IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

In re Patent Application: John Colgrave	**Confirmation No.:** 5696
Application No.: 10/561,260	**Examiner:** Joshua A. Murdough
Filed: October 16, 2006	
Title: "User Access to a Registry of Business Entity Definitions"	**Group Art Unit:** 3621

Mail Stop Appeal Brief-Patents
Commissioner for Patents
P.O. Box 1450
Alexandria, VA 22313-1450

TRANSMITTAL OF APPEAL BRIEF

Sir:

Transmitted herewith is the Appeal Brief in this application with respect to the Notice of Appeal filed on **October 6, 2010**.

(**X**) The fee for filing this Appeal Brief is **$540.00** (37 CFR 41.20).

( ) No Additional Fee Required.

(complete (a) or (b) as applicable)

The proceedings herein are for a patent application and the provision of 37 CFR 1.136 (a) apply.

( ) (a) Applicant petitions for an extension of time under 37 CFR 1.136 (fees: CFR 1.17(a)-(d)) for the total number of months checked below:

( )	one month	$130.00
( )	two months	$490.00
( )	three months	$1110.00
( )	four months	$1730.00

( ) The extension fee has already been filed in this application

(**X**) (b) Applicant believes that no extension of time is required. However, this conditional petition is being made to provide for the possibility that applicant had inadvertently overlooked the need for a petition and fee for extension of time.

Please charge to Deposit Account 09-0461/ GB920030007US1 the sum of $540.00. At any time during the pendency of this application, please charge any fees required or credit any over payment to Deposit Account 09-0461/ GB920030007US1 pursuant to 37 CFR 1.25. Additionally please charge any fees to Deposit Account 09-0461/ GB920030007US1 under CFR 1.16 through 1.21 inclusive, and any other section in the Title 37 of the Code of Federal Regulations that may regulate fees.

Respectfully submitted,

By: /Steven L. Nichols/
Steven L. Nichols (Reg. No.: 40,326)
Attorney/Agent for Applicant(s)
Telephone No.: (801) 237-0251
Date: December 2, 2010

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

In the Patent Application of John Colgrave	Group Art Unit: 3621
Application No. 10/561,260	Examiner: Joshua A. Murdough
Filed: October 16, 2006	Confirmation No.: 5696
For: User Access to a Registry of Business Entity Definitions	

APPEAL BRIEF

Mail Stop Appeal Brief - Patents
Commissioner for Patents
P.O. Box 1450
Alexandria, VA 22313-1450

Sir:

This is an Appeal Brief under Rule 41.37 appealing the decision of the Primary Examiner dated August 4, 2010 (the "final Office Action" or "Action"). Each of the topics required by Rule 41.37 is presented herewith and is labeled appropriately.

I. Real Party in Interest

The inventors have assigned their interests in the present application to International Business Machines Corporation ("IBM"), which has a principal place of business at New Orchard Road, Armonk, NY 10504. Accordingly, the real party in interest is IBM.

II. Related Appeals and Interferences

There are no appeals or interferences related to the present application of which the Appellant is aware.

III. Status of Claims

Claims 1-16 have been previously cancelled without prejudice or disclaimer.

Claims 17-30 are pending in the application and stand finally rejected.

Accordingly, Appellant appeals from the final rejection of claims 17-30, which claims are presented in the Appendix.

IV. Status of Amendments

No amendments have been filed subsequent to the Office Action of August 4, 2010, from which Appellant takes this appeal.

V. Summary of Claimed Subject Matter

A summary is given below of the subject matter recited in each of the independent claims at appeal. The citation to passages in the specification and drawings for each claim element does not imply that the limitations from the cited passages in the specification and drawings should be read into the corresponding claim elements. *See Superguide Corp. v. DirecTV Enterprises, Inc.*, 358 F.3d 870, 875, 69 USPQ2d 1865, 1868 (Fed. Cir. 2004); M.P.E.P. § 2111.01(II). Citations to one or more line numbers in connection with a specific paragraph refer to the lines within that paragraph, and not necessarily to the lines in the page where that paragraph appears.

Turning to Appellant's specific claims,

Claim 17 recites:

A method for a registry of business entity definitions to handle user requests to access business entity definitions, the method comprising:

receiving a request (511, Fig. 5) in a processor (101, Fig. 1) associated with said registry (302, Fig. 5) from a user to access a business entity definition (201, Fig. 4) (*e.g.*, Appellant's specification, p. 5 lines 21-37, p. 10 lines 28-13, p. 11 lines 32-34, p. 12 lines 24-33, Figs. 4-6) comprising a plurality of information elements (201, 203, 206, Fig. 4) (*e.g.*, Appellant's specification, p. 8 line 6 to p. 9 line 37, Figs. 4-6), each of said information elements (201, 203, 206, Fig. 4) having permission details (401, 402, 404, Fig. 4) associated therewith (*e.g.*, Appellant's specification, p. 8 line 6 to p. 9 line 37, Figs. 4-6);

obtaining the identity of the user from data associated with the request (511, Fig. 5) with said processor (101, Fig. 1) (*e.g.*, Appellant's specification, p. 10 lines 35-37, p. 11 lines 35-37, Figs. 4-6);

determining with said processor (101, Fig. 1), for each of said information elements (201, 203, 206, Fig. 4), whether the user has permission to access that said information element (201, 203, 206, Fig. 4) based on the permission details (401, 402, 404, Fig. 4) associated with that said information element (201, 203, 206, Fig. 4) and said identity of the user (*e.g.*, Appellant's specification, p. 8 line 6 to p. 9 line 37, Figs. 4-6); and

with said processor (101, Fig. 1), denying the user access to those information elements (201, 203, 206, Fig. 4) for which it is determined that the user does not have permission (*e.g.*, Appellant's specification, p. 8 line 6 to p. 9 line 37, Figs. 4-6).

Claim 21 recites:

A registry (302, Fig. 3) of business entity definitions, the registry comprising:

at least one processor (101, Fig. 1) (*e.g.*, Appellant's specification, p. 5 lines 21-37, p. 12 lines 24-33, Fig. 1); and

a computer readable memory (102, 103, Fig. 1) communicatively coupled to said processor (101, Fig. 1) having said business entity definitions stored thereon (*e.g.*, Appellant's specification, p. 5 lines 21-37, p. 12 lines 24-33, Fig. 1);

wherein said processor (101, Fig. 1) is configured to:

receive a request (511, Fig. 5) from a user to access a business entity definition (201, Fig. 4) comprising a plurality of information elements (201, 203, 206, Fig. 4) (*e.g.*, Appellant's specification, p. 8 line 6 to p. 9 line 37, Figs. 4-6), each of said information elements (201, 203, 206, Fig. 4) having permission details (401, 402, 404, Fig. 4) associated therewith (*e.g.*, Appellant's specification, p. 8 line 6 to p. 9 line 37, Figs. 4-6);

obtain the identity of the user from data associated with the request (511, Fig. 5) (*e.g.*, Appellant's specification, p. 10 lines 35-37, p. 11 lines 35-37, Figs. 4-6);

determine, for each of said information elements (201, 203, 206, Fig. 4), whether the user has permission to access that said information element (201, 203, 206, Fig. 4) based on the permission details (401, 402, 404, Fig. 4) associated with that said information element (201, 203, 206, Fig. 4) and said identity of the user (*e.g.*, Appellant's specification, p. 8 line 6 to p. 9 line 37, Figs. 4-6); and

deny the user access to those information elements (201, 203, 206, Fig. 4) for which it is determined that the user does not have permission (*e.g.*, Appellant's specification, p. 8 line 6 to p. 9 line 37, Figs. 4-6).

Claim 27 recites:

A computer program product for a registry of business entity definitions to handle user requests to access business entity definitions, the computer program product comprising:

a computer readable storage medium (102, 103, Fig. 1) having computer usable program code embodied thereon (*e.g.*, Appellant's specification, p. 5 lines 21-37, p. 12 lines 24-33, Fig. 1), the computer usable program code comprising:

computer usable program code configured to receive a request (511, Fig. 5) from a user to access a business entity definition (201, Fig. 4) comprising a plurality of information elements (201, 203, 206, Fig. 4) (*e.g.*, Appellant's specification, p. 8 line 6 to p. 9 line 37, Figs. 4-6), each of said information elements (201, 203, 206, Fig. 4) having permission details (401, 402, 404, Fig. 4) associated therewith (*e.g.*, Appellant's specification, p. 8 line 6 to p. 9 line 37, Figs. 4-6);

computer usable program code configured to obtain the identity of the user from data associated with the request (511, Fig. 5) (*e.g.*, Appellant's specification, p. 10 lines 35-37, p. 11 lines 35-37, Figs. 4-6);

computer usable program code configured to determine, for each of said information elements (201, 203, 206, Fig. 4), whether the user has permission to access that said information element (201, 203, 206, Fig. 4) based on the permission details (401, 402, 404, Fig. 4) associated with that said information element (201, 203, 206, Fig. 4) and said identity of the user (*e.g.*, Appellant's specification, p. 8 line 6 to p. 9 line 37, Figs. 4-6); and

computer usable program code configured to deny the user access to those information elements (201, 203, 206, Fig. 4) for which it is determined that the user does not have permission (*e.g.*, Appellant's specification, p. 8 line 6 to p. 9 line 37, Figs. 4-6).

VI. Grounds of Rejection to be Reviewed on Appeal

The Office Action raised the following grounds of rejection.

(1) Claims 21-26 stand rejected under 35 U.S.C. § 112, first paragraph, as allegedly failing to comply with the written description requirement.

(2) Claims 21-26 stand rejected under 35 U.S.C. § 112, first paragraph, allegedly failing to comply with the enablement requirement.

(3) Claims 17-30 stand rejected under 35 U.S.C. § 112, second paragraph, as being allegedly indefinite.

(4) Claims 17, 21, and 27 stand rejected under 35 U.S.C. § 102(b) as being allegedly anticipated by U.S. Patent No. 5,414,852 to Kramer et al. ("Kramer").

(5) Claims 19-20, 23-24, and 29-30 stand rejected under 35 U.S.C. § 103(a) as being allegedly obvious over Kramer in view of U.S. Patent No. 5,629,980 to Stefik et al. ("Stefik").

(6) Claims 18, 22, 25-26, and 28 stand rejected under 35 U.S.C. § 103(a) as being allegedly obvious over Kramer in view of U.S. Patent Application Publication No. 2002/0046157 by Solomon ("Solomon").

(7) Claims 17, 21, and 27 stand alternately rejected under 35 U.S.C. § 103(a) as being allegedly obvious over Kramer in view of Stefik.

Accordingly, Appellant hereby requests review of each of these grounds of rejection in the present appeal.

VII. Argument

The claims do not stand or fall together. Instead, Appellant presents separate arguments for various independent and dependent claims. Each of these arguments is set forth below with separate headings and subheadings as required by 37 C.F.R. § 41.37(c)(1)(vii).

(1) Claims 21-26 comply with the written description requirement of 35 U.S.C. § 112, first paragraph.

Claims 21-26 stand rejected under 35 U.S.C. § 112, first paragraph, as allegedly failing to meet the written description requirement. For at least the following reasons, this rejection is improper and should be reversed.

A. "At least one Processor"

According to the final Office Action, claim 21 fails to meet the written description requirement of 35 U.S.C. § 112, first paragraph because it recites a registry comprising "at least one processor," where the specification only teaches a single-processor embodiment. (Action, p. 2). The Examiner asserts that "[b]ecause Appellants did not disclose multiple processors in their original disclosure, the limitation "at least one processor," which claims registries with more than one processor, is new matter." (*Id.*). This assertion is erroneous on multiple levels.

In the first place, Appellant notes that the claimed term "at least one processor" does not require multiple processors, as the Examiner asserts. A registry with only a single processor is a registry with "at least one processor." (Claim 21). As such, Appellant's specification and drawings clearly teach a registry system with "at least one processor." (*Id.*; Appellant's specification, p. 5 line 21 to p. 7 line 13, p. 10 line 28 to p. 12 line 33, Figs. 1-6).

The Manual for Patenting Examination and Procedure (M.P.E.P.) has embraced the tests of the Federal Circuit for determining whether subject matter complies with the written description requirement. *See* M.P.E.P. § 2163.03. For example, a claim complies with the written description requirement if "the description clearly allow[s] persons of ordinary skill in the art to recognize that he or she invented what is claimed." *In re Gosteli*, 872 F.2d 1008, 1012, 10 USPQ2d 1614, 1618 (Fed. Cir. 1989); *id.* (quoting *Gosteli*).

More recent decisions by the Federal Circuit follow the same vein. In *Ariad Pharmaceuticals, Inc. v. Eli Lilly and Co.*, the Federal Circuit states that

> the hallmark of written description is disclosure . . . the test requires an objective inquiry into the four corners of the specification from the perspective of a person of ordinary skill in the art. Based on that inquiry, the specification must describe an invention understandable to that skilled artisan and show that the inventor actually invented the invention claimed.

598 F.3d 1336, 1351 (Fed. Cir. 2010) (*en banc*).

Applying this standard to claim 21, it will be readily apparent that the specification describes what is claimed, namely a registry of business entity definitions that comprises "at least one processor." (Claim 21; Appellant's specification, p. 5 line 21 to p. 7 line 13, p. 10 line 28 to p. 12 line 33, Figs. 1-6).

Moreover, the Examiner incorrectly interprets 35 U.S.C. § 112, first paragraph as imposing a requirement that the specification explicitly describe **all possible embodiments** of a claim. On this subject the relevant case law is quite clear. The court in *Ariad* expressly stated that "[t]he written description requirement does not demand either examples or an actual reduction to practice." *Ariad*, 598 F.3d at 1352. Additionally, it is long-settled that "[a] patent need not teach, and preferably omits, what is well known in the art." *Spectra-Physics, Inc. v. Coherent, Inc.*, 827 F.2d 1524, 1536, 3 USPQ2d 1737, 1745 (Fed. Cir. 1987). Applying these considerations to the present rejection of claim 21, the Examiner has obviously erred in requiring Appellant's specification to expressly describe embodiments in which a registry is implemented by more than one processor. (*See* Action, p. 2). There can be no dispute that it has been long known in the art that multiple processors can and are used to perform tasks performed by a single processor. Thus, 35 U.S.C. § 112, first paragraph imposes no requirement that the specification explicitly describe a registry implemented by multiple processors for claim 21 to comply with the written description requirement.

B. Registry comprising Processor

The Action additionally asserts that Appellant's specification describes only a UDDI registry that "may be installed on [a] client/server" and that "[b]ecause the registry is installed on the client/server it cannot be the client/server." (Action, pp. 2-3). Following this vein, the Examiner concludes that "[b]ecause a registry with a processor is not necessarily present in the original disclosure, it is new matter." (*Id.*). Appellant strongly disagrees.

Appellant's specification describes the UDDI registry as an active component that performs certain functionality. (*See, e.g.*, Appellant's specification, p. 10 line 32 to p. 11 line 13). This functionality cannot be performed by merely software alone, as software is merely a set of instructions. Rather, the functions of the UDDI registry must inherently be performed by a machine executing software. Thus, the UDDI registry described in Appellant's specification inherently has a machine component. Taken in context with this fact, the specification's statement that "a UDDI registry may be installed on any such client/server" plainly teaches that the described client/server provides the essential hardware component to implement UDDI registry functionality. (Appellant's specification, p. 5 lines 23-24). As such, the specification plainly teaches a UDDI registry that "comprises" a processor.

C. Registry comprising Computer Readable Memory

For similar reasons, the Action further asserts that Appellant's specification does not describe a registry comprising a computer readable memory. (Action, p. 3; *see* claim 21). Appellant strongly disagrees.

Again, Appellant's specification describes the UDDI registry as an active component that performs certain functionality. (*See, e.g.*, Appellant's specification, p. 10 line 32 to p. 11 line 13). This functionality cannot be performed by merely software alone, as software is merely a set of instructions. Rather, the functions of the UDDI registry must inherently be performed by a machine executing software from a computer-readable memory. Thus, the UDDI registry described in Appellant's specification inherently has a machine component. Taken in context with this fact, the specification's statement that "a UDDI registry may be installed on any such client/server" plainly teaches that the described client/server provides the essential hardware components to implement UDDI registry functionality. (Appellant's specification, p. 5 lines 23-24). Among the hardware described in the client/server of Appellant's specification are "a RAM volatile memory element" and a "non-volatile memory." (Appellant's specification, p. 5 lines 27-29, Fig. 1; *see also* p. 12 lines 24-33). As such, the specification plainly teaches a UDDI registry that "comprises" a computer readable memory.

D. Conclusion

For at least the reasons given above, the Examiner has failed to establish that any part of claim 21 fails to comply with the written description requirement of 35 U.S.C. § 112, first paragraph. Consequently, the Final Office Action fails to meet its requisite burden of establishing the *prima facie* unpatentability of claim 21. Therefore, the rejection of claim 21 and its dependent claims based on the written description requirement of 35 U.S.C. § 112, first paragraph should be reversed.

<u>(2) Claims 21-26 comply with the enablement requirement of 35 U.S.C. § 112, first paragraph.</u>

Claims 21-26 stand rejected under 35 U.S.C. § 112, first paragraph, as allegedly failing to comply with the enablement requirement of 35 U.S.C. § 112, first paragraph. For at least the following reasons, this rejection is improper and should be reversed.

A. All Structures for Performing Claimed Functions

Claim 21 recites a processor that is "configured to" perform various tasks, including "receive a request," "obtain the identity of [a] user," "determine . . . whether the user has permission," and "deny the user access" to certain information elements. The Examiner concludes that because the processor is modified by "purely functional" limitations, "the processor is modified by pure functional language with no limitation of structure," and that claim 21 is therefore directed to "*all* structures for performing the claimed function." (Action, pp. 3-4). Because, the Examiner argues, the specification does not disclose all possible structures for performing the claimed function, claim 21 does not comply with the enablement requirement of 35 U.S.C. § 112, second paragraph. (*Id.*) (citing to *Ex Parte Miyazaki*, 89 USPQ2d 1207, 1215-1217 (BPAI 2008)). Appellant strongly disagrees.

The Examiner's assertion that the functionality performed by the processor recited in claim 21 is not tied to any underlying structure is perplexing. Claim 21 explicitly recites that a processor, an inherently structural element, performs all of the functionality described in claim 21. The processor performs this functionality as exemplified in Appellant's specification at, for example, p. 10 line 28 to p. 13 line 7, and Figs. 4-6. Thus, unlike the patent application in the *Miyazaki* cased cited by the Examiner, claim 21 does not recite any "purely functional claim element." (*See Miyazaki*, 89 USPQ2d at 1215-1217; Action, pp. 3-4).

The teachings of Appellant's specification are sufficient to meet the enablement requirement for all of the functionality performed by the processor of claim 21. *See N. Telecom, Inc. v. Datapoint Corp.*, 908 F.2d 931, 941 (Fed. Cir. 1990) (computer-implemented software is enabled when "a programmer of reasonable skill could write a satisfactory program" implementing the functionality).

B. Importing Structure from the specification

The Action further asserts that no structure can be imported from the specification to remedy what the Examiner perceives as a lack of structural recitations in claim 21. In response, Appellant respectfully notes that even if such importation were permissible, which it is not, it would not be necessary. The term "processor" is already a structural limitation on each element of functionality recited in claim 21.

C. Purely Software Configuration

The Action further asserts that "Appellants have enabled only a software configuration in their original disclosure." (Action, p. 4) (citing to Appellant's specification, p. 12 lines 24-33). Thus, according to the Examiner, because the "claim encompasses a hardware configuration for performing the receiving step," claim 21 must be rejected for failing to meet the enablement requirement. Appellant strongly disagrees.

Again, Appellant's specification describes the UDDI registry as an active component that performs certain functionality. (*See, e.g.*, Appellant's specification, p. 10 line 32 to p. 11 line 13). This functionality cannot be performed by merely software alone, as software is merely a set of instructions. Rather, the functions of the UDDI registry must inherently be performed by a machine executing software. Thus, the UDDI registry described in Appellant's specification necessarily has a machine component. Taken in context with this fact, the specification's statement that "a UDDI registry may be installed on any such client/server" plainly teaches that the described client/server provides the essential hardware component to implement UDDI registry functionality. (Appellant's specification, p. 5 lines 23-24). As such, the specification plainly enables a hardware configuration for performing each element of functionality in claim 21.

D. Conclusion

For at least the reasons given above, the Examiner has failed to establish that any part of claim 21 fails to comply with the enablement requirement of 35 U.S.C. § 112, first paragraph. Consequently, the Final Office Action fails to meet its requisite burden of establishing the *prima facie* unpatentability of claim 21. Therefore, the rejection of claim 21 and its dependent claims based on the enablement requirement of 35 U.S.C. § 112, first paragraph should be reversed.

(3) Claims 17-30 comply with 35 U.S.C. § 112, second paragraph.

Claims 17-30 stand rejected under 35 U.S.C. § 112, second paragraph, as being allegedly indefinite. For at least the following reasons, this rejection is improper and should be reversed.

A. "Permission Details"

Claims 17, 21, and 27 recite different forms of "receiving a request in a processor associated with said registry from a user to access a business entity definition comprising a plurality of information elements, each of said information elements having permission details associated therewith." (Claim 17). According to the Final Office Action, "[o]ne of ordinary skill in the art would not understand what the 'permission details' are associated with." (Action, p. 5). In particular, the Action asserts that it is unclear whether "the information elements have permission details and the permission details are associated with the request" or whether "the permission details are associated with the processor." (*Id.*).

Appellant strongly disagrees. Claims 17, 21, and 27 could not be clearer in their recitation of "information elements *having permission details associated therewith*." (*E.g.*, claim 17) (emphasis added). The language of these claims plainly indicates that the permission details are associated with the recited information elements. Furthermore, the independent claims further indicate that the permission details associated with each information element indicate whether a given user has permission to access that information element. (Claims 17, 21, 27) (*e.g.*, "determining . . . whether the user has permission to access that said information element based on the permission details associated with that said information element and said identity of the user."). The specification describes the use of permission details in this context. (*See e.g.*, Appellant's specification, p. 3 line 25 to p. 4 line 16, p. 8 line 6 to p. 9 line 32, p. 10 line 1 to p. 10 line 25, Figs. 4-6).

The Examiner's ability to reject claims in a pending patent application under 35 U.S.C. § 112, second paragraph is governed by the requirements of the M.P.E.P. According to M.P.E.P. § 2173.02,

> [t]he essential inquiry pertaining to [the requirement for definiteness of 35 U.S.C. § 112, second paragraph] is whether the claims set out and circumscribe a particular subject matter with a reasonable degree of clarity and particularity. Definiteness of claim language must be analyzed, not in a vacuum, but in light of:
>
> (A) The content of the particular application disclosure;
> (B) The teachings of the prior art; and
> (C) The claim interpretation that would be given by one possessing the ordinary level of skill in the pertinent art at the time the invention was made.

Applying this standard, we find that the content of the particular application disclosure and the claim interpretation that would be given by one possessing ordinary skill in the art at the time of the invention clearly indicate the scope of the term "permission details" recited in claims 17, 21, and 27. As such, the Examiner's position concerning the definiteness of the term "permission details" is misplaced and improper. Consequently, the term "permission details" is not indefinite under 35 U.S.C. § 112, second paragraph.

B. "Plurality of Information Elements"

The Action further asserts with respect to claims 17, 21, and 27 that it is unclear whether "the request or the business entity definition" comprises the recited "plurality of information elements." (Action, p. 5). Appellant again strongly disagrees.

The specification describes business entity definitions which include a plurality of information elements, but does not describe a request that includes a plurality of information elements. (*See, e.g.*, Appellant's specification, p. 3 line 14 to p. 4 line 24, p. 8 lines 6-10, p. 9 line 34 to p. 10 line 11, p. 10 line 41 to p. 11 line 2, p. 11 line 40 to p. 12 line 4, p. 12 lines 13-18, p. 12 line 40 to p. 13 line 7, Figs. 2-4). Thus, the disclosure of the application unquestionably supports the interpretation that the "business entity definition" is the element that comprises the "plurality of information elements." (Claim 17).

Furthermore, the independent claims recite "receiving a request in a processor associated with said registry from a user to access ***a business entity definition comprising a plurality of information elements***." A reading of this phrase under the conventions of the English language results in the clear interpretation that the "business entity definition" is what comprises the "plurality of information elements." (Claim 17). Because a person having ordinary skill in the art at the time of the invention would read English phrases in a conventional way, such a person would also interpret the phrase as reading that the recited "business entity definition" is what comprises "the plurality of information elements." (*Id.*).

Again, the M.P.E.P. dictates that a determination of the definiteness of a claim term under 35 U.S.C. § 112, second paragraph requires a consideration of

> (A) The content of the particular application disclosure;
> (B) The teachings of the prior art; and
> (C) The claim interpretation that would be given by one possessing the ordinary level of skill in the pertinent art at the time the invention was made.

M.P.E.P. § 2173.02. Under this analysis, it becomes readily apparent that the content of the application disclosure and the claim interpretation that would be made by one having ordinary skill in the art support a unified, definite interpretation of the independent claims as indicating that a business entity definition comprises a plurality of information elements. (*E.g.*, claim 17). As such, the Examiner's position concerning the definiteness of the term "permission details" is misplaced and improper. The phrase "receiving a request in a processor associated with said registry from a user to access a business entity definition comprising a plurality of information elements" is not indefinite under 35 U.S.C. § 112, second paragraph.

C. "Business Entity Definition"

Claims 17-30 also stand rejected as being indefinite under 35 U.S.C. § 112, second paragraph, because (A) Examiner has not found a lexicographic definition for the phrase "business entity definition" and (B) it is the Examiner's position that this phrase is "not known to those of ordinary skill in the art." (Action, p. 6). This rejection is utterly improper and inappropriate, as the Examiner has not set forth adequate grounds of rejecting these claims.

With respect to the Examiner's failure to find a lexicographic definition of the term "business entity definition" in the specification, the M.P.E.P. is quite clear that "a claim term that is not used ***or defined*** in the specification is not indefinite if the meaning of the claim term is discernible." M.P.E.P. § 2173.02 (citing to *Bancorp Svcs., L.L.C. v. Hartford Life Ins. Co.*, 359 F.3d 1367, 1372, 69 USPQ2d 1996, 1999-2000 (Fed. Cir. 2004)). The Examiner's requirement that each claim term be either specifically defined in the specification or well-known in the art creates a false dichotomy which precludes the notion that a claim's term can be discernible from how it is used in the specification in spite of the lack of a formal definition. *See id.*

In determining whether a claim term is indefinite under § 112, second paragraph, the M.P.E.P. requires an examination of the "content of the particular application disclosure." M.P.E.P. § 2173.02. The "broadest reasonable interpretation" given to claim terms during examination must be "consistent with the specification." *Phillips v. AWH Corp.*, 415 F.3d 1303, 75 USPQ2d 1321 (Fed. Cir. 2005) (quoting *In re Am. Acad. of Sci. Tech. Ctr.*, 367 F.3d 1359, 1364, 70 USPQ2d 1827 (Fed. Cir. 2004)); *see* M.P.E.P. § 2111 (quoting *Phillips*). Therefore, if the specification makes the meaning of a claimed term clear, that term is plainly not indefinite under 35 U.S.C. § 112, second paragraph.

The term "business entity definition" is used throughout Appellant's specification as an entity in a registry, such as a Universal Description Discovery and Integration (UDDI) registry, through which a business entity publishes information about the services it provides in the form of definitions, each definition including one or more information elements. (*See, e.g.*, Appellant's specification, title, p. 2 lines 1-4, p. 3 lines 5-9, p. 4 lines 18-24). The specification specifically states the following:

> The preferred embodiment is discussed in terms of a business entity definition and a UDDI registry according to the UDDI specification. The UDDI specification defines a set of XML schemas which are templates for creating definitions of business entities. As a result a business entity definition may be an instance of one or more these [sic] defined XML schemas.

(*Id.*, p. 5 lines 33-37).

Appellant's Declaration of October 5, 2010 provides multiple documents known in the art prior to the filing date of the present application. These documents outline the specifications of the known UDDI registry protocol described in the specification. In particular, the UDDI specification describes a registry protocol for a businessEntity data structure which "serves as the top-level information manager for all of the information about a particular set of information related to a business unit. (*See, e.g.*, Appellant's Declaration of Oct. 5, 2010, Ex. A p. 6). The elements of the businessEntity data structure are described in detail in these documents, and are consistent with what is described in Appellant's specification. (*See, e.g.*, *id.*, Ex. B pp. 3-6).

The specification further makes it clear that in a UDDI embodiment, the "business entity definition" comprises "a businessEntity data structure 201 which includes descriptive information 202 about the business and information about one or more business services which the business entity offers." (Appellant's specification, p. 5 line 42 to p. 6 line 3; *see also* p. 6 lines 28-30, p. 6 line 35 to p. 7 line 15). The specification also states that "while the invention has been described in terms of a UDDI registry a skilled person would realise that a registry could be implemented, which is not a UDDI registry, but provides a similar function to a UDDI registry. (*Id.*, p. 12 lines 35-38).

Accordingly, one having ordinary skill in the art would read the specification as teaching that a business entity definition is a definition that includes either a UDDI businessEntity data structure or a non-UDDI data structure that performs a similar function to a UDDI businessEntity data structure. Because the composition and use of UDDI businessEntity data structures are well-known in the art, one having ordinary skill in the art would readily understand the scope of the term "business entity definition" in this context. (*See generally* Appellant's Declaration of Oct. 5, 2010).

The Action asserts that because only claim 26 is expressly directed to a UDDI registry, the knowledge and use of business entity definitions in the context of UDDI registries has no bearing on whether the term "business entity definition" is indefinite. (Action, pp. 18-19). Appellant strongly disagrees. Again, the specification is clear that the term "business entity definition" relates both to definitions which include a UDDI businessEntity data structure and to definitions which include a non-UDDI data structure that is analogous to the businessEntity data structure. (Appellant's specification, p. 5 lines 33-37, p. 5 line 42 to p. 6 line 3, p. 6 lines 28-30, p. 6 line 35 to p. 7 line 15). Thus, knowledge of UDDI businessEntity data structures in the relevant art sufficiently renders the more generic "business entity definition" term definite in the context of the specification.

The Action further asserts that "it is not clear [in claims 17, 21, and 27] whether any 'entity' is part of a registry ('entity in a registry') or if it modifies the business ('business entity')." (Action, p. 18). Appellant strongly disagrees. The conventions of the English language do not permit reading the term "entity" as modifying the term "registry" in the independent claims. For example, claim 17 recites "receiving a request in a processor associated with said registry from a user to access *a business entity definition comprising a plurality of information elements*." (Emphasis added). Claims 21 and 27 recite essentially this same phrase. No one having ordinary skill in the art would interpret the term "entity" in the phrase as modifying the term "registry" instead of the term "business." By asserting otherwise, the Examiner unfairly and unreasonably imposes an invented ambiguity as a basis for rejecting the instant claims as indefinite.

The Action also objects to the Appellant's use of Federal Circuit cases in responding to the present rejection. In the Examiner's view, the Board's decision in *Ex parte Miyazaki* renders moot any definiteness standard used for issued patent claims in *inter parte* proceedings because the instant claims have not yet been issued. (Action, p. 17) (citing to *Ex parte Miyazaki*, 89 USPQ2d 1207, 1211 (BPAI 2008) (precedential). Appellant strongly disagrees. The Federal Circuit holdings cited in the present Brief can all be found in the M.P.E.P. in sections relevant to the Examiner's rejections made under 35 U.S.C. § 112, second paragraph. *See, e.g.*, M.P.E.P. §§ 2111, 2173. Consequently, by adopting these standards from *inter parte* case law in the M.P.E.P., the USPTO has exercised its prerogative to include certain *inter parte* standards from the Federal Circuit in the "lower threshold of ambiguity" used by the USPTO personnel during the examination of claims for compliance with the definiteness requirement. *Miyazaki*, 89 USPQ2d at 1211.

The Action further cites to *Miyazaki* for its holding that "if a claim is amenable to two or more plausible claim constructions, the USPTO is justified in . . . holding the claim unpatentable under 35 U.S.C. § 112, second paragraph, as indefinite." (Action, p. 17) (citing to *id.*). Appellant is aware of this holding. Nevertheless, the Examiner has not met this threshold, because the Examiner has not been able to demonstrate that the term "business entity definition" is amenable to any definition other than what would be logically inferred from the claims and specification of the present application in view of the ordinary skill of the art.

For at least the above reasons, the meaning of the term "business entity definition" is not indefinite under 35 U.S.C. § 112, second paragraph, because this term sets out and circumscribes a particular subject matter with a reasonable degree of clarity and particularity." M.P.E.P. § 2173.01. The scope of the subject matter is apparent in light of (A) the "content of the particular application disclosure," (B) the "teachings of the prior art," and (C) "the claim interpretation that would be given by one possessing the ordinary level of skill in the pertinent art at the time the invention was made," as required by the M.P.E.P. *Id.* Consequently, the rejection of claims 17-30 under 35 U.S.C. § 112, second paragraph based on the term "business entity definition" should be reversed.

D. "Registry of Business Entity Definitions"

The Action further rejects claim 21 as indefinite under 35 U.S.C. § 112, second paragraph, for the recitation of a "registry of business entity definitions, the registry comprising: at least one processor; and a computer readable memory." (Action, p. 6). In this regard, the Examiner asserts that one of ordinary skill in the art would recognize a "registry of business entity definitions" as an arrangement of data, and that the addition of "physical components" renders the term "registry of business entity definitions" entirely incomprehensible to those having ordinary skill in the art. Appellant strongly disagrees.

As described above, the specification describes the UDDI registry as an active component that performs certain functionality. (*See, e.g.*, Appellant's specification, p. 10 line 32 to p. 11 line 13). This functionality cannot be performed by a mere arrangement of data. Rather, the functions of the UDDI registry must inherently be performed by a machine executing software. Thus, the UDDI registry described in Appellant's specification inherently has a machine component. Taken in context with this fact, the scope of the term "registry of business entity definitions" would be readily understood by one having ordinary skill in the art as including a processor. As such, the specification plainly teaches a UDDI registry that "comprises" a processor.

Even if *arguendo* the registry described in the specification did not inherently require hardware, a person having ordinary skill in the art of Web Services and registries would easily ascertain the scope of claim 21. Persons skilled in the art of Web Services and registries would certainly have sufficient intelligence and reasoning capacity to determine that the recitation of a registry comprising a processor and a memory would entail a processor that manages registry data stored on the memory.

For at least the above reasons, the meaning of the term "business entity definition" is not indefinite under 35 U.S.C. § 112, second paragraph, because this term sets out and circumscribes a particular subject matter with a reasonable degree of clarity and particularity." M.P.E.P. § 2173.01. The scope of the subject matter is apparent in light of (A) the "content of the particular application disclosure," (B) the "teachings of the prior art," and (C) "the claim interpretation that would be given by one possessing the ordinary level of skill in the pertinent art at the time the invention was made," as required by the M.P.E.P. *Id.*

E. "Business Service Entity Information Element,"

Claim 25 (incorrectly identified as claim 26 in the Action) stands further rejected under 35 U.S.C. § 112, second paragraph, because of the Examiner's position that the term "business service entity information element," is indefinite. Appellant strongly disagrees.

Again, the Examiner is inappropriately requiring that these terms be either specifically defined in the specification or well-known in the art to comply with § 112, second paragraph. For at least the reasons given above with respect to the definiteness of the term "business service entity," this standard has no legal basis.

The correct inquiry pertaining to compliance with 35 U.S.C. § 112, second paragraph is "whether the claims set out and circumscribe a particular subject matter with a reasonable degree of clarity and particularity" in light of the "(A) [t]he content of the particular application disclosure; (B) [t]he teachings of the prior art; and (C) [t]he claim interpretation that would be given by one possessing the ordinary level of skill in the pertinent art at the time the invention was made." M.P.E.P. § 2173.01. No explicit definition of these terms is required for the "content of the particular application disclosure" to provide a basis for whether the claimed subject matter has "a reasonable degree of clarity and particularity." *Id.*; *see Bancorp*, 359 F.3d at 1372, 69 USPQ2d 1999-2000 (a claim term this is not defined in the specification is not indefinite if the meaning of the claim term is discernible) (quoted in M.P.E.P. § 2173.01).

The Examiner has failed to comprehend that claim 25 is directed to a UDDI registry, and the terms recited in claim 25 reflect concepts that are well-known in the field of UDDI. (*See, e.g.*, Appellant's Declaration of Oct. 5, 2010, Exs. A-D). For example, Exhibit A of Appellant's Declaration describes a "businessService" entity as one of the "core information elements that make up the UDDI information model." (*Id.* at Ex. A, p. 11; *see also* pp. 5-7). Furthermore, Appellant's specification describes the term "business service information element" in context, from which one having ordinary skill in the art could easily determine the metes and bounds of the subject matter claimed. (Appellant's Specification, p. 6 line 11 to p. 7 line 15, p. 8 line 6 to p. 10 line 22).

As such, the Examiner has rejected claim 25 while clearly failing to perform the necessary inquiry into whether the terms "whether the claims set out and circumscribe a particular subject matter with a reasonable degree of clarity and particularity" in light of the "(A) [t]he content of the particular application disclosure; (B) [t]he teachings of the prior art; and (C) [t]he claim interpretation that would be given by one possessing the ordinary level of skill in the pertinent art at the time the invention was made." M.P.E.P. § 2173.01. As demonstrated above, the term "business service information element" is well-defined in the language of the claim, the application disclosure, and the teachings of the prior art. Thus, the term “technical model information element” does not render claim 25 indefinite under 35 U.S.C. § 112, second paragraph.

F. “Binding Template Information Element”

Claim 25 (incorrectly identified as claim 26 in the Action) stands further rejected under 35 U.S.C. § 112, second paragraph, because of the Examiner’s position that the term “binding template information element,” is indefinite. Appellant strongly disagrees.

Again, the Examiner is inappropriately requiring that these terms be either specifically defined in the specification or well-known in the art to comply with § 112, second paragraph. For at least the reasons given above with respect to the definiteness of the term “business service entity,” this standard has no legal basis.

The correct inquiry pertaining to compliance with 35 U.S.C. § 112, second paragraph is “whether the claims set out and circumscribe a particular subject matter with a reasonable degree of clarity and particularity” in light of the “(A) [t]he content of the particular application disclosure; (B) [t]he teachings of the prior art; and (C) [t]he claim interpretation that would be given by one possessing the ordinary level of skill in the pertinent art at the time the invention was made.” M.P.E.P. § 2173.01. No explicit definition of these terms is required for the “content of the particular application disclosure” to provide a basis for whether the claimed subject matter has “a reasonable degree of clarity and particularity.” *Id.*; *see Bancorp*, 359 F.3d at 1372, 69 USPQ2d 1999-2000 (a claim term this is not defined in the specification is not indefinite if the meaning of the claim term is discernible) (quoted in M.P.E.P. § 2173.01).

The Examiner has failed to comprehend that claim 25 is directed to a UDDI registry, and the terms recited in claim 25 reflect concepts that are well-known in the field of UDDI.

(*See, e.g.*, Appellant's Declaration of Oct. 5, 2010, Exs. A-D). For example, Exhibit A of Appellant's Declaration describes a "bindingTemplate" entity as one of the "core information elements that make up the UDDI information model." (*Id.* at Ex. A, p. 11; *see also* pp. 5-7). Furthermore, Appellant's specification describes the term "binding template information element" in context, from which one having ordinary skill in the art could easily determine the metes and bounds of the subject matter claimed. (Appellant's Specification, p. 6 line 11 to p. 7 line 15, p. 8 line 6 to p. 10 line 22).

As such, the Examiner has rejected claim 25 while clearly failing to perform the necessary inquiry into whether the terms "whether the claims set out and circumscribe a particular subject matter with a reasonable degree of clarity and particularity" in light of the "(A) [t]he content of the particular application disclosure; (B) [t]he teachings of the prior art; and (C) [t]he claim interpretation that would be given by one possessing the ordinary level of skill in the pertinent art at the time the invention was made." M.P.E.P. § 2173.01. As demonstrated above, the term "binding template information element" is well-defined in the language of the claim, the application disclosure, and the teachings of the prior art. Thus, the term "technical model information element" does not render claim 25 indefinite under 35 U.S.C. § 112, second paragraph.

G. "Technical Model Information Element"

Claim 25 (incorrectly identified as claim 26 in the Action) stands further rejected under 35 U.S.C. § 112, second paragraph, because of the Examiner's position that the term "technical model information element," is indefinite. Appellant strongly disagrees.

Again, the Examiner is inappropriately requiring that these terms be either specifically defined in the specification or well-known in the art to comply with § 112, second paragraph. For at least the reasons given above with respect to the definiteness of the term "business service entity," this standard has no legal basis.

The correct inquiry pertaining to compliance with 35 U.S.C. § 112, second paragraph is "whether the claims set out and circumscribe a particular subject matter with a reasonable degree of clarity and particularity" in light of the "(A) [t]he content of the particular application disclosure; (B) [t]he teachings of the prior art; and (C) [t]he claim interpretation that would be given by one possessing the ordinary level of skill in the pertinent art at the time the invention was made." M.P.E.P. § 2173.01. No explicit definition of these terms is required for the "content of the particular application disclosure" to provide a basis for whether the claimed subject matter has "a reasonable degree of clarity and particularity." *Id.*; *see Bancorp*, 359 F.3d at 1372, 69 USPQ2d 1999-2000 (a claim term this is not defined in the specification is not indefinite if the meaning of the claim term is discernible) (quoted in M.P.E.P. § 2173.01).

The Examiner has failed to comprehend that claim 25 is directed to a UDDI registry, and the terms recited in claim 25 reflect concepts that are well-known in the field of UDDI. (*See, e.g.*, Appellant's Declaration of Oct. 5, 2010, Exs. A-D). For example, Exhibit A of Appellant's Declaration describes a "tModel" entity which describes a "technical fingerprint of a Web Service" entity as one of the "core information elements that make up the UDDI information model." (*Id.* at Ex. A, pp. 7, 11). Furthermore, Appellant's specification equates the term "technical model" functionally with the UDDI tModel entity, and describes the entity in a way that one having ordinary skill in the art could easily determine the metes and bounds of the subject matter claimed. (Appellant's Specification, p. 6 line 11 to p. 7 line 15, p. 8 line 6 to p. 10 line 22).

As such, the Examiner has rejected claim 25 while clearly failing to perform the necessary inquiry into whether the terms "whether the claims set out and circumscribe a particular subject matter with a reasonable degree of clarity and particularity" in light of the "(A) [t]he content of the particular application disclosure; (B) [t]he teachings of the prior art; and (C) [t]he claim interpretation that would be given by one possessing the ordinary level of skill in the pertinent art at the time the invention was made." M.P.E.P. § 2173.01. As demonstrated above, the term "technical model information element" is well-defined in the language of the claim, the application disclosure, and the teachings of the prior art. Thus, the term "technical model information element" does not render claim 25 indefinite under 35 U.S.C. § 112, second paragraph.

H. "Business Entity Information Element"

Claim 25 (incorrectly identified as claim 26 in the Action) stands further rejected under 35 U.S.C. § 112, second paragraph, because of the Examiner's position that the term "business service entity information element," is indefinite. Appellant strongly disagrees.

Again, the Examiner is inappropriately requiring that these terms be either specifically defined in the specification or well-known in the art to comply with § 112, second paragraph. For at least the reasons given above with respect to the definiteness of the term "business service entity," this standard has no legal basis.

The correct inquiry pertaining to compliance with 35 U.S.C. § 112, second paragraph is "whether the claims set out and circumscribe a particular subject matter with a reasonable degree of clarity and particularity" in light of the "(A) [t]he content of the particular application disclosure; (B) [t]he teachings of the prior art; and (C) [t]he claim interpretation that would be given by one possessing the ordinary level of skill in the pertinent art at the time the invention was made." M.P.E.P. § 2173.01. No explicit definition of these terms is required for the "content of the particular application disclosure" to provide a basis for whether the claimed subject matter has "a reasonable degree of clarity and particularity." *Id.*; *see Bancorp*, 359 F.3d at 1372, 69 USPQ2d 1999-2000 (a claim term this is not defined in the specification is not indefinite if the meaning of the claim term is discernible) (quoted in M.P.E.P. § 2173.01).

The Examiner has failed to comprehend that claim 25 is directed to a UDDI registry, and the terms recited in claim 25 reflect concepts that are well-known in the field of UDDI. (*See, e.g.*, Appellant's Declaration of Oct. 5, 2010, Exs. A-D). For example, Exhibit A of Appellant's Declaration describes a "businessEntity" entity as one of the "core information elements that make up the UDDI information model." (*Id.* at Ex. A, p. 11; *see also* pp. 5-7). Furthermore, Appellant's specification describes the term "business entity information element" in context, from which one having ordinary skill in the art could easily determine the metes and bounds of the subject matter claimed. (Appellant's Specification, p. 6 line 11 to p. 7 line 15, p. 8 line 6 to p. 10 line 22).

As such, the Examiner has rejected claim 25 while clearly failing to perform the necessary inquiry into whether the terms "whether the claims set out and circumscribe a particular subject matter with a reasonable degree of clarity and particularity" in light of the "(A) [t]he content of the particular application disclosure; (B) [t]he teachings of the prior art; and (C) [t]he claim interpretation that would be given by one possessing the ordinary level of skill in the pertinent art at the time the invention was made." M.P.E.P. § 2173.01. As demonstrated above, the term "technical model information element" is well-defined in the language of the claim, the application disclosure, and the teachings of the prior art. Thus, the term "technical model information element" does not render claim 25 indefinite under 35 U.S.C. § 112, second paragraph.

I. Conclusion

For at least the above reasons, the Action has not met the Office's burden to establish the *prima facie* failure to comply with 35 U.S.C. § 112, second paragraph, of any of claims 17-30. Therefore, the rejection of claims 17-30 on these grounds is improper and should be reversed.

<u>(4) Claims 17, 21, and 27 are patentable over Kramer.</u>

Claims 17, 21, and 27 stand rejected under 35 U.S.C. § 102(b) as being allegedly anticipated by Kramer. For at least the following reasons, this rejection is improper and should be reversed.

<u>Claim 17:</u>

Claim 17 recites:

> A method for a registry of business entity definitions to handle user requests to access business entity definitions, the method comprising:
> receiving a request in a processor associated with said registry from a user to access ***a business entity definition comprising a plurality of information elements, each of said information elements having permission details associated therewith***;
> obtaining the identity of the user from data associated with the request with said processor;
> determining with said processor, ***for each of said information elements, whether the user has permission to access that said information element based on the permission details associated with that said information element and said identity of the user***; and
> with said processor, ***denying the user access to those information elements for which it is determined that the user does not have permission***.

(Emphasis added).

Appellant notes that "[t]he examiner bears the initial burden . . . of presenting a *prima facie* case of unpatentability." *In re Oetiker*, 977 F.2d 1443, 1445, 24 USPQ2d 1443, 1444 (Fed. Cir. 1992). In a rejection made under § 102, this burden is substantial, as a *prima facie* case of anticipation requires a demonstration that "*each and every element* as set forth in the claim is found, either expressly or inherently described, in a single prior art reference." *Verdegaal Bros. v. Union Oil Co. of California*, 814 F.2d 628, 631, 2 U.S.P.Q.2d 1051, 1053 (Fed. Cir. 1987) (emphasis added); *see* M.P.E.P. § 2131.

The Federal Circuit has further clarified that "unless a reference discloses within the four corners of the document *not only all of the limitations claimed but also all of the limitations arranged or combined in the same way as recited in the claim*, it cannot be said to prove prior invention of the thing claimed and, thus, cannot anticipate under 35 U.S.C. § 102." *Net MoneyIN, Inc. v. Verisign, Inc.*, 545 F.3d 1359, 1371, 88 U.S.P.Q.2d 1751, 1759 (Fed. Cir. 2008) (emphasis added).

In light of these considerations, the recent Office Action does not meet the requisite burden to establish the *prima facie* anticipation of claim 17. Specifically, the Examiner has failed to demonstrate that the Kramer reference teaches or suggests *each and every element* recited in claim 17. *See Verdegaal*, 814 F.2d at 631, 2 USPQ2d at 1053.

Kramer is directed to "a data processing system [that] includes a plurality of data objects which are accessible by application programs through a system level interface." (Kramer, col. 1 lines 56-59). According to Kramer, "[e]ach data object has an associated user access list" and "at least one key indicating which applications can access that object." (*Id.*, lines 59-62). "Both the application identifier key and the user who invoked that application must match the identifier information in the data object for access to be allowed to that object." (*Id.*, lines 64-67).

The Action cites to teachings in Kramer about a user attempting to access such a data object, asserting that the data object of Kramer reads on the "business entity definition" recited in claim 17. (Action, pp. 9-10) (citing to Kramer, col. 3 lines 50-51). Appellant respectfully disagrees. In construing a claim, the meaning of words used in the claims is determined by the plain meaning of the words in light of the meaning given to those words in the specification. *Markman v. Westview Instruments*, 116 U.S. 1384, 38 USPQ2d 1461 (1996); *McGill, Inc. v. John Zink Co.*, 736 F.2d 666, 674 (Fed. Cir. 1984); *ZMI Corp. v. Cardiac Resuscitator Corp.*, 884 F.2d 1576, 1580, 6 USPQ2d 1557, 1560-61 (Fed. Cir. 1988) ("words must be used in the same way in both the claims and the specification").

The plain meaning of the term "business entity definition" implies that data structure that defines a specific business-related entity. Additionally, the specification uses the term "business entity definition" to refer to an organization of data that "provides details of services provided . . . by a business" and "descriptive information . . . of the business." (Appellant's specification, p. 5 line 40 to p. 6 line 9). Thus, from the plain language of the term "business entity definition" and the fact that the specification only uses this term to refer to data defining a specific business entity, to anticipate the term "business entity definition" the data object of Kramer must at least be associated with and define a business entity. Kramer does not teach or suggest anywhere that each of its data objects is associated with a business entity. Accordingly, the data object of Kramer does not read on the "business entity definition" recited in claim 17, and Kramer does not teach or suggest "receiving a request from a user to access a business entity definition." (Claim 17).

As evidence that Kramer teaches the plurality of information elements recited in claim 17, the Action cites to Kramer's teaching that "[w]hen access is requested of a particular data object . . . through the system level interface, the system level interface compares the identifier of the original invoking user and the calling data managers with those stored in the access list 36 and key list 38. (Action, p. 10) (citing to Kramer, col. 4 lines 66-68). According to the Examiner, the "identifier of the original invoking user and the calling data managers" read on the "information elements" of the business entity definition in claim 17. This position is logically indefensible. Kramer is clear that "[w]hen a user invokes a data manager application, either directly or indirectly, a user identifier is contained within the process identifier of a process associated with such invocation," and that "when a data manager makes a call to the system level interface the identity of the calling data manager is made available to the system level interface as well." (Kramer, col. 3 lines 20-34). As such, Kramer leaves no doubt that the identifiers of the user and the calling data managers are not a part of the data object that is being accessed. While Appellant expressly states that Kramer does not teach or suggest a business entity definition, even if *arguendo* Kramer did teach a business entity definition, the Examiner has still failed to demonstrate any teaching in Kramer of a business entity definition "comprising a plurality of information elements." (Claim 17). Indeed, Kramer does not teach or suggest such "a business entity definition comprising a plurality of information elements" anywhere. (*Id.*).

Additionally, by asserting that the identifier of the user and the identifier of the calling data managers read on the "information elements having permission details associated therewith" recited in claim 17, the Action puts itself in a logically untenable position. The problem with this position stems from the fact that claim 17 recites "determining . . . for each of said information elements, whether the user has permission to access that said information element based on the permission details associated with that said information element." (Claim 17).

The Action further asserts that the access control data stored for a data object reads on the permission details recited in claim 17. Thus, under the Examiner's interpretation of Kramer, if a user wishes to access a data object, permissions associated with the data object will be compared to an identifier of the user and an identifier of a calling manager to determine whether the user may access his or her own identifier and the identifier of the calling manager. Kramer plainly does not teach such a system. Even if it did, such a system would not read on that of claim 17.

Finally, Kramer does not teach or suggest "denying the user access to those information elements for which it is determined that the user does not have permission." (Claim 17). Again, claim 17 is explicit that a business entity definition includes multiple information elements and that a user is denied access on an individual basis to those information elements for which the user does not have permission. In this regard, the Action cites to Kramer's teaching of using access control lists to grant or deny a user access to a data object based on the user's identity and the identity of a calling manager. (Action, p. 10) (citing to Fig. 4). However, in asserting that this subject matter in Kramer reads on claim 17, the Examiner forgets his previous assertion that the identifier of the user and the identifier of the calling manager read on the information elements of claim 17. Yet, Kramer does not teach or suggest any type of controlled access to these identifiers. Rather, Kramer only teaches controlled access to a stored singular data object. Accordingly, the Action has not demonstrated that Kramer teaches or suggests "denying the user access to those information elements for which it is determined that the user does not have permission." (Claim 17).

Again, "[a] claim is anticipated [under 35 U.S.C. § 102] only if *each and every element* as set forth in the claim is found, either expressly or inherently described, in a single prior art reference." *Verdegaal*, 814 F.2d at 631, 2 U.S.P.Q.2d at 1053; *see Verisign*, 545 F.3d at 1371, 88 U.S.P.Q.2d at 1759; M.P.E.P. § 2131. Thus, Kramer cannot anticipate claim 17 because, for the above reasons, Kramer fails to teach or suggest all of the subject matter present in claim 17. Because the Office has not met its burden to demonstrate the *prima facie* unpatentability of claim 17, the rejection of claim 17 and its dependent claims based on Kramer should be reversed.

Claim 21:

Claim 21 recites:

> A registry of business entity definitions, the registry comprising:
> at least one processor; and
> a computer readable memory communicatively coupled to said processor having said business entity definitions stored thereon;
> wherein said processor is configured to:
> ***receive a request from a user to access a business entity definition comprising a plurality of information elements, each of said information elements having permission details associated therewith***;
> obtain the identity of the user from data associated with the request;
> ***determine, for each of said information elements, whether the user has permission to access that said information element based on the permission details associated with that said information element and said identity of the user***; and
> ***deny the user access to those information elements for which it is determined that the user does not have permission.***

(Emphasis added).

Kramer also fails to anticipate the registry of business entity definitions of claim 21 because Kramer does not teach or suggest all of the subject matter recited in claim 21. Specifically, as amply demonstrated above with respect to independent claim 17, Kramer does not teach or suggest "receiv[ing] a request from a user to access a business entity definition comprising a plurality of information elements, each of said information elements having permission details associated therewith," "determin[ing], for each of said information elements, whether the user has permission to access that said information element based on the permission details associated with that said information element and said identity of the user;" or "deny[ing] the user access to those information elements for which it is determined that the user does not have permission." (Claim 21).

Again, "[a] claim is anticipated [under 35 U.S.C. § 102] only if *each and every element* as set forth in the claim is found, either expressly or inherently described, in a single prior art reference." *Verdegaal*, 814 F.2d at 631, 2 U.S.P.Q.2d at 1053; *see Verisign*, 545 F.3d at 1371, 88 U.S.P.Q.2d at 1759; M.P.E.P. § 2131. Thus, Kramer cannot anticipate claim 21 because, for the above reasons, Kramer fails to teach or suggest all of the subject matter present in claim 21. Because the Office has not met its burden to demonstrate the *prima facie* unpatentability of claim 21, the rejection of claim 21 and its dependent claims based on Kramer should be reconsidered and withdrawn.

Claim 27:

Claim 27 recites:

> A computer program product for a registry of business entity definitions to handle user requests to access business entity definitions, the computer program product comprising:
> a computer readable storage medium having computer usable program code embodied thereon, the computer usable program code comprising:
> computer usable program code configured to ***receive a request from a user to access a business entity definition comprising a plurality of information elements, each of said information elements having permission details associated therewith***;
> computer usable program code configured to obtain the identity of the user from data associated with the request;
> computer usable program code configured to ***determine, for each of said information elements, whether the user has permission to access that said information element based on the permission details associated with that said information element and said identity of the user***; and
> computer usable program code configured to ***deny the user access to those information elements for which it is determined that the user does not have permission***.

(Emphasis added).

Kramer also fails to anticipate the computer program product of claim 27 because Kramer does not teach or suggest all of the subject matter recited in claim 27. Specifically, as amply demonstrated above with respect to independent claim 17, Kramer does not teach or suggest "receiv[ing] a request from a user to access a business entity definition comprising a plurality of information elements, each of said information elements having permission details associated therewith," "determin[ing], for each of said information elements, whether the user has permission to access that said information element based on the permission details associated with that said information element and said identity of the user;" or "deny[ing] the user access to those information elements for which it is determined that the user does not have permission." (Claim 27).

Again, "[a] claim is anticipated [under 35 U.S.C. § 102] only if *each and every element* as set forth in the claim is found, either expressly or inherently described, in a single prior art reference." *Verdegaal*, 814 F.2d at 631, 2 U.S.P.Q.2d at 1053; *see Verisign*, 545 F.3d at 1371, 88 U.S.P.Q.2d at 1759; M.P.E.P. § 2131. Thus, Kramer cannot anticipate claim 27 because, for the above reasons, Kramer fails to teach or suggest all of the subject matter present in claim 27. Because the Office has not met its burden to demonstrate the *prima facie* unpatentability of claim 27, the rejection of claim 27 and its dependent claims based on Kramer should be reconsidered and withdrawn.

(5) Claims 19-20, 23-24, and 29-30 are patentable over Kramer in view of Stefik.

Claims 19-20, 23-24, and 29-30 stand rejected under 35 U.S.C. § 103(a) as being allegedly obvious over Kramer in view of Stefik. This rejection is improper and should be reversed at least for the same reasons given above in favor of the patentability of independent claims 17, 21, and 27. *See In re Fine*, 837 F.2d 1071, 1076, 5 USPQ2d 1596 (Fed. Cir. 1988) (if an independent claim is nonobvious, then any claim depending therefrom is nonobvious); M.P.E.P. § 2143.03.

<u>(6) Claims 18, 22, 25-26, and 28 are patentable over Kramer in view of Solomon.</u>

Claims 18, 22, 25-26, and 28 stand rejected under 35 U.S.C. § 103(a) as being allegedly obvious over Kramer in view of Solomon. This rejection is improper and should be reversed at least for the same reasons given above in favor of the patentability of independent claims 17, 21, and 27. *See Fine*, 837 F.2d at 1076 (if an independent claim is nonobvious, then any claim depending therefrom is nonobvious); M.P.E.P. § 2143.03.

<u>(7) Claims 17, 21, and 27 are patentable over Kramer in view of Stefik.</u>

Claims 17, 21, and 27 stand rejected under 35 U.S.C. § 103(a) as being allegedly obvious over Kramer in view of Stefik. For at least the following reasons, this rejection is improper and should be reversed.

Claim 17 recites:

> A method for a registry of business entity definitions to handle user requests to access business entity definitions, the method comprising:
> receiving a request in a processor associated with said registry from a user to access ***a business entity definition comprising a plurality of information elements, each of said information elements having permission details associated therewith***;
> obtaining the identity of the user from data associated with the request with said processor;
> determining with said processor, ***for each of said information elements, whether the user has permission to access that said information element based on the permission details associated with that said information element and said identity of the user***; and
> with said processor, ***denying the user access to those information elements for which it is determined that the user does not have permission***.

(Emphasis added).

Claim 21 recites:

> A registry of business entity definitions, the registry comprising:
> at least one processor; and
> a computer readable memory communicatively coupled to said processor having said business entity definitions stored thereon;
> wherein said processor is configured to:
> ***receive a request from a user to access a business entity definition comprising a plurality of information elements, each of said information elements having permission details associated therewith***;
> obtain the identity of the user from data associated with the request;
> ***determine, for each of said information elements, whether the user has permission to access that said information element based on the permission details associated with that said information element and said identity of the user***; and
> ***deny the user access to those information elements for which it is determined that the user does not have permission***.

(Emphasis added).

Claim 27 recites:

> A computer program product for a registry of business entity definitions to handle user requests to access business entity definitions, the computer program product comprising:
> a computer readable storage medium having computer usable program code embodied thereon, the computer usable program code comprising:
> computer usable program code configured to ***receive a request from a user to access a business entity definition comprising a plurality of information elements, each of said information elements having permission details associated therewith***;
> computer usable program code configured to obtain the identity of the user from data associated with the request;
> computer usable program code configured to ***determine, for each of said information elements, whether the user has permission to access that said information element based on the permission details associated with that said information element and said identity of the user***; and
> computer usable program code configured to ***deny the user access to those information elements for which it is determined that the user does not have permission***.

(Emphasis added).

In rejecting claims 17, 21, and 27 under these alternate grounds, the Examiner maintains the assertion that Kramer teaches "receiving a request . . . from a user to access a business entity definition comprising a plurality of information elements, each of said information elements having permission details associated therewith" and "determining . . . for each of said information elements, whether the user has permission to access that said information element based on the permission details associated with that said information element and said identity of the user;" and "denying the user access to those information elements for which it is determined that the user does not have permission" for the same reasons used by the Examiner to reject these claims under § 102(b). (Action, pp. 13-14) (quoting claim 17). Appellant strongly disagrees with these assertions for the same reasons given above with respect to the § 102(b) rejection of claims 17, 21, and 27, noting that the Examiner has utterly failed to establish that Kramer teaches or suggests any of this subject matter.

Stefik does not remedy the deficiencies of Kramer in this regard. Specifically, Stefik utterly fails to teach or suggest at least the subject matter of "receiving a request . . . from a user to access a business entity definition comprising a plurality of information elements, each of said information elements having permission details associated therewith" and "determining . . . for each of said information elements, whether the user has permission to access that said information element based on the permission details associated with that said information element and said identity of the user." (*See* claims 17, 21, and 27). Notably, the Action only relies on Stefik for the teaching of a processor. (Action, p. 14).

According to the Supreme Court, the factual inquiries set forth in *Graham v. John Deere Co.*, 383 U.S. 1, 17-18 (1966), "continue to define the inquiry that controls" obviousness rejections under 35 U.S.C. § 103. *KSR Int'l v. Teleflex Inc.*, 550 U.S. 398, 407 (2007). Under the analysis required by *Graham* to support a rejection under 35 U.S.C. § 103,

> the scope and content of the prior art are to be determined; differences between the prior art and the claims at issue are to be ascertained; and the level of ordinary skill in

> the pertinent art resolved. Against this background, the obviousness or nonobviousness of the subject matter is determined.

Graham, 383 U.S. at 17-18.

While these inquiries are factual, the ultimate determination of obviousness is a conclusion of law made in view of the totality of the resolved *Graham* factors. *KSR*, 550 U.S. at 427; *Graham*, 383 U.S. at 17.

Applying the *Graham* analysis to the present rejection of claims 17, 21, and 27, the scope and content of the prior art, as evidenced by Kramer and Stefik, does not include all of the claimed subject matter, including at least "receiving a request . . . from a user to access a business entity definition comprising a plurality of information elements, each of said information elements having permission details associated therewith" and "determining . . . for each of said information elements, whether the user has permission to access that said information element based on the permission details associated with that said information element and said identity of the user." (*See* claims 17, 21, and 27).

The differences between the cited prior art and claims 17, 21, and 27 are significant because the claimed subject matter provides features and advantages not known or available in the cited prior art. Therefore, no combination of the cited prior art references would feasibly produce the subject matter recited in claims 17, 21, and 27. As such, no one having ordinary skill in the art at the time of the invention would have been able to combine the Kramer and Stefik references to arrive at the claimed subject matter as a whole.

This fact is significant because under the rationale chosen by the Examiner to reject claims 17, 21, and 27, a *prima facie* case of obviousness requires a factual demonstration that "the prior art included each element claimed, although not necessarily in a single prior art reference, **with the only difference between the claimed invention and the prior art being**

the lack of actual combination of the elements in a single prior art reference." M.P.E.P. § 2143(A) (emphasis added).

Consequently, the cited prior art will not support a rejection of claims 17, 21, and 27 under 35 U.S.C. § 103 and *Graham*, and the Office has not met its requisite burden to establish the *prima facie* obviousness of claims 17, 21, and 27. For at least these reasons, the rejection of claims 17, 21, and 27 should be reversed/reconsidered and withdrawn.

In view of the foregoing, it is submitted that the final rejection of the pending claims is improper and should not be sustained. Therefore, a reversal of the Rejection of August 4, 2010 is respectfully requested.

Respectfully submitted,

[illegible]

DATE: December 2, 2010

Steven L. Nichols
Registration No. 40,326

Steven L. Nichols, Esq.
Van Cott, Bagley, Cornwall & McCarthy
36 South State Street
Suite 1900
Salt Lake City, Utah 84111

(801) 237-0251 (phone)
(801) 237-0871 (fax)

VIII. CLAIMS APPENDIX

1-16. (canceled)

17. (previously presented) A method for a registry of business entity definitions to handle user requests to access business entity definitions, the method comprising:

receiving a request in a processor associated with said registry from a user to access a business entity definition comprising a plurality of information elements, each of said information elements having permission details associated therewith;

obtaining the identity of the user from data associated with the request with said processor;

determining with said processor, for each of said information elements, whether the user has permission to access that said information element based on the permission details associated with that said information element and said identity of the user; and

with said processor, denying the user access to those information elements for which it is determined that the user does not have permission.

.

18. (previously presented) The method of claim 17 wherein the request specifies a search criteria, and wherein the method further comprises:

determining with said processor what information in the business entity definition the user wishes to access using the search criteria to locate the business entity definition; and

determining with said processor whether the user has permission to access the information that the user wishes to access.

19. (previously presented) The method of claim 17, wherein determining with said processor whether the user has permission to access information in the business entity definition from permission details associated with the business entity definition and the identity of the user comprises determining with said processor whether a user has permission to access an information element from an access policy and permission details associated with a different information element.

20. (previously presented) The method of claim 19 wherein the information elements are in a hierarchy and wherein determining with said processor whether the user has permission to access information in the business entity definition from permission details associated with the business entity definition and the identity of the user comprises determining with said processor that a user does not have permission to access a first information element if permission details associated with one or more second information elements directly beneath the first information element in the hierarchy indicate that user does not have access to one or more of the second information elements.

21. (previously presented) A registry of business entity definitions, the registry comprising:

at least one processor; and

a computer readable memory communicatively coupled to said processor having said business entity definitions stored thereon;

wherein said processor is configured to:

receive a request from a user to access a business entity definition comprising a plurality of information elements, each of said information elements having permission details associated therewith;

obtain the identity of the user from data associated with the request;

determine, for each of said information elements, whether the user has permission to access that said information element based on the permission details associated with that said information element and said identity of the user; and

deny the user access to those information elements for which it is determined that the user does not have permission.

.

22. (previously presented) The registry of claim 21 wherein the request specifies a search criteria, and wherein the data processing host device is further configured to:

determine what information in the business entity definition the user wishes to access using the search criteria to locate the business entity definition.

23. (previously presented) The registry of claim 21, wherein the the data processing host device is further configured to determine whether a user has permission to access an information element from an access policy and permission details associated with a different information element.

24. (previously presented) The registry of claim 23 wherein the information elements are in a hierarchy and the access policy specifies that a user does not have permission to access a first information element if permission details associated with one or more second information elements directly beneath the first information element in the

hierarchy indicate that user does not have access to one or more of the second information elements.

25. (previously presented) The registry of claim 21, wherein the data processing host device is further configured to:

locate the permission details in a file system in which the permission details are in a location in the file system which is defined according to the information element with which they are associated.

26. (previously presented) The registry of claim 21, wherein the registry is a UDDI registry and the information in the business entity definition is a business entity information element, the business entity information element containing one or more business service entity information elements, each business service entity containing one or more binding template information elements and each binding template containing one or more references each referring to a technical model information element.

27. (previously presented) A computer program product for a registry of business entity definitions to handle user requests to access business entity definitions, the computer program product comprising:

a computer readable storage medium having computer usable program code embodied thereon, the computer usable program code comprising:

computer usable program code configured to receive a request from a user to access a business entity definition comprising a plurality of information elements, each of said information elements having permission details associated therewith;

computer usable program code configured to obtain the identity of the user from data associated with the request;

computer usable program code configured to determine, for each of said information elements, whether the user has permission to access that said information element based on the permission details associated with that said information element and said identity of the user; and

computer usable program code configured to deny the user access to those information elements for which it is determined that the user does not have permission.

28. (previously presented) The computer program product of claim 27 wherein the request specifies a search criteria, and further comprising:

computer usable program code configured to determine what information in the business entity definition the user wishes to access using the search criteria to locate the business entity definition; and

computer usable program code configured to determine whether the user has permission to access the information that the user wishes to access.

29. (previously presented) The computer program product of claim 27, wherein the computer usable program code configured to determine whether the user has permission to access information in the business entity definition from permission details associated with the business entity definition and the identity of the user comprises computer usable program code configured to determine whether a user has permission to access an information element from an access policy and permission details associated with a different information element.

30. (previously presented) The computer program product of claim 30 wherein the information elements are in a hierarchy and wherein determining whether the user has permission to access information in the business entity definition from permission details associated with the business entity definition and the identity of the user comprises determining that a user does not have permission to access a first information element if permission details associated with one or more second information elements directly beneath the first information element in the hierarchy indicate that user does not have access to one or more of the second information elements.

IX. Evidence Appendix

The following pages include a copy of a Declaration filed by Appellant on Oct. 5, 2010, including Exhibits, the Declaration being referenced in section VII of the present Appeal Brief.

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

In the Patent Application of

John Colgrave

Application No. 10/561,260

Filed: **October 16, 2006**

For: User Access to a Registry of Business Entity Definitions

Group Art Unit: 3621

Examiner: Joshua A. Murdough

Confirmation No.: 5696

DECLARATION UNDER 37 C.F.R. § 1.132

Mail Stop Appeal Brief - Patents
Commissioner for Patents
P.O. Box 1450
Alexandria, VA 22313-1450

Sir:

I declare the following:

1. I am an attorney of record in the prosecution of the instant application, U.S. Patent Application Serial No. 10/561,260.

2. Attached to the present declaration as Exhibit A is a document entitled "UDDI Technical White Paper," which is a known documentation of the UDDI standard. This document was published as early as September 6, 2000, and is presently available on the Internet at http://uddi.org/pubs/Iru_UDDI_Technical_White_Paper.pdf.

3. Attached to the present declaration as Exhibit B is a document entitled "Modeling the UDDI Schema with UML," which describes a known XML Schema associated with the UDDI Specification. This document was published as early as February, 2001, and is presently available on the Internet at http://www.xmlmodeling.com/models/uddi/article/ModelingUDDI.pdf.

4. Attached to the present declaration as Exhibit C is a document entitled "UDDI Programmer's API 1.0," which is a known documentation of the UDDI standard. This document was published as early as June 28, 2002, and is presently available on the Internet at http://uddi.org/pubs/ProgrammersAPI-V1.01-Published-20020628.pdf.

5. Attached to the present declaration as Exhibit D is a document entitled "UDDI Version 2.03 Data Structure Reference," which is a known documentation of the UDDI standard. This document was published as early as July 19, 2002, and is presently available on the Internet at http://uddi.org/pubs/DataStructure_v2.pdf.

6. Each of Exhibits A-D was publicly available at the time the present application was filed.

7. Exhibit A describes a data structure known as a "businessEntity element," of "businessEntity," which "serves as the top-level information manager for all of the information about a particular set of information related to a business unit." (Exhibit A, p. 5). Exhibits B-D also refer to the businessEntity element, provide additional descriptions, uses, and specifications for the data structure. (*See, e.g.*, Exhibit B, pp. 3-5; Exhibit C, pp. 12-14, 17, Exhibit D, pp. 5-6, 8-14).

8. These disclosures demonstrate that the terms "businessEntity," "businessEntity element," and their equivalents had accepted and understood meanings in the relevant art at the time the present patent application was filed.

9. I hereby declare that all statements made herein of my own knowledge are true and that all statements made on information and belief are believed to be true; and further that these statements were made with the knowledge that willful false statements and the like so made are punishable by fine or imprisonment, or both, under Section 1001 of Title 18 of the United States Code and that such willful false statements may jeopardize the validity of the application or any patent issued thereon.

[illegible]

Steven L. Nichols
Attorney for Applicant

October 5, 2010
Date

Exhibit A

uddi.org

Universal Description, Discovery and Integration

UDDI Technical White Paper

September 6, 2000

Abstract

Universal Description, Discovery and Integration (UDDI) is a specification for distributed Web-based information registries of Web services. UDDI is also a publicly accessible set of implementations of the specification that allow businesses to register information about the Web services they offer so that other businesses can find them.

Web services are the next step in the evolution of the World Wide Web (WWW) and allow programmable elements to be placed on Web sites where others can access distributed behaviors. UDDI registries are used to promote and discover these distributed Web services. This paper describes the capabilities that these registries add to the World Wide Web.

The intended audience is the anyone looking for a conceptual overview of UDDI for the purpose of understanding what it is, who uses it, and how a distributed registry makes it possible for your programs to discover and interact with Web services that other companies expose on the Web

Introduction

Overview

The Universal Description, Discovery and Integration (UDDI) specifications define a way to publish and discover information about Web services. The term "Web service" describes specific business functionality exposed by a company, usually through an Internet connection, for the purpose of providing a way for another company or software program to use the service.

Web services are becoming the programmatic backbone for electronic commerce. For example, one company calls another's service to send a purchase order directly via an Internet connection. Another example is a service that calculates the cost of shipping a package of a certain size or weight, so many miles via a specific carrier.

At first glance, it would seem simple to manage the process of Web service *discovery*. After all, if a known business partner has a known electronic commerce gateway, what's left to discover? The tacit assumption, however, is that all of the information is already known. When you want to find out which business partners have which services, the ability to discover the answers can quickly become difficult. One option is to call each partner on the phone, and then try to find the right person to talk with. For a business that is exposing Web services, having to staff enough highly technical people to satisfy random discovery demand is difficult to justify.

Another way to solve this problem is through an approach that uses a Web services description file on each company's Web site. After all, Web crawlers work by accessing a registered URL and are able to discover and index text found on nests of Web pages. The "robots.txt" approach, however, is dependent on the ability for a crawler to locate each Web site and the location of the service description file on that Web site. This distributed approach is potentially scalable but lacks a mechanism to insure consistency in service description formats and for the easy tracking of changes as they occur.

UDDI takes an approach that relies upon a distributed registry of businesses and their service descriptions implemented in a common XML format.

UDDI business registrations and the UDDI business registry

The core component of the UDDI project is the UDDI business registration, an XML file used to describe a business entity and its Web services. Conceptually, the information provided in a UDDI business registration consists of three components: "white pages" including address, contact, and known identifiers; "yellow pages" including industrial categorizations based on standard taxonomies; and "green pages", the technical information about services that are exposed by the business. Green pages include references to specifications for Web services, as well as support for pointers to various file and URL based discovery mechanisms if required.

Using UDDI

UDDI includes the shared operation of a business registry on the Web. For the most part, programs and programmers use the UDDI Business Registry to locate information about services and, in the case of programmers, to prepare systems that are compatible with advertised Web services or to describe their own Web services for others to call. The UDDI Business Registry can be used at a business level to check whether a given partner has particular Web service interfaces, to find companies in a given industry with a given type of service, and to locate information about how a partner or intended partner has exposed a Web service in order to learn the technical details required to interact with that service.

After reading this paper, the reader will have a clearer understanding of the capabilities defined in the UDDI specifications and have a clearer understanding of the role of Web service registries that implement these specifications.

Background

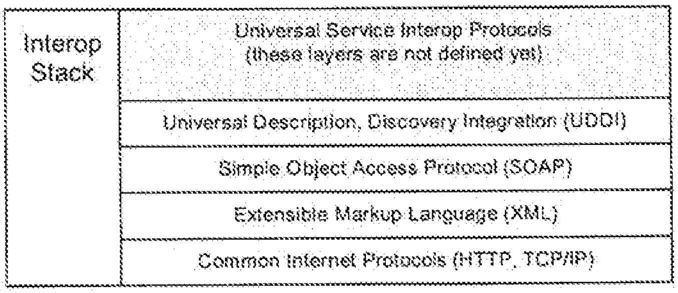
The number of ways that companies are using the World Wide Web varies considerably. Many companies are starting to define ways to allow their internal applications to interact with the business systems at other companies using the emerging Web infrastructure. Left alone, each company invents a unique approach based on the experiences of designers, available technologies, and project budgets. The proliferation of integration approaches and unique solutions have spawned an entire sub-industry focused on bridging incompatible service layers within and across company boundaries.

Recent work within the W3C starts to raise hopes that Extensible Markup Language (XML) will play a role in simplifying the exchange of business data between companies. Further, collaboration between computer industry giants and small companies alike have outlined a framework called SOAP that allows one program to invoke service interfaces across the Internet, without the need to share a common programming language or distributed object infrastructure. All of this is good news for companies feeling the cost pressures associated with electronic commerce because the foundations for common interoperability standards are being laid. Because of these foundation technologies and emerging standards, some of the intractable problems of the past are becoming easier to approach.

From XML and SOAP, one can observe that the integration and interoperability problem has been simplified in layers. XML provides a cross-platform approach to data encoding and formatting. SOAP, which is built on XML, defines a simple way to package information for exchange across system boundaries. SOAP bindings for HTTP are built on this packaging protocol and define a way to make remote procedure calls between systems in a manner that is independent of the programming language or operating system choices made by individual companies. Prior approaches involved complex distributed object standards or technology bridging software. Neither of these approaches has proven to be cost effective in the long run. Using XML and SOAP, this cross-language, cross-platform approach simplifies the problem of making systems at two companies compatible with each other.

Even when one considers XML and SOAP, though, there are still vast gaps through which any two companies can fall in implementing a communications infrastructure. As any industry pundit will tell you: "What is required is a full end-to-end solution, based on standards that are universally supported on every computing platform." Clearly, there is more work to do to achieve this goal. The UDDI specifications borrow the lesson learned from XML and SOAP to define a next-layer-up that lets two companies share a way to query each other's capabilities and to describe their own capabilities.

The following diagram depicts this layered view:


Interop Stack
Universal Service Interop Protocols (these layers are not defined yet)
Universal Description, Discovery Integration (UDDI)
Simple Object Access Protocol (SOAP)
Extensible Markup Language (XML)
Common Internet Protocols (HTTP, TCP/IP)

UDDI is a "next layer" in an emerging *stack* enabling rich Web services. UDDI uses standards-based technologies such as TCP/IP, HTTP, XML and SOAP to create a uniform service description format and service discovery protocol.

Figure 1

UDDI – the technical discovery layer

The Universal Description, Discovery and Integration (UDDI) specification describes a conceptual cloud of Web services and a programmatic interface that define a simple framework for describing any kind of Web service. The specification consists of several related documents and an XML schema that defines a SOAP-based programming protocol for registering and discovering Web services. These specifications were defined over a series of months by technicians and managers from several leading companies. Together, these companies have undertaken the task of building the first implementation of the UDDI services and running these services as a publicly accessible, multi-site partnership that shares all registered information.

The following diagram shows the relationship between the specifications, the XML schema and the UDDI business registry cloud that provides "register once, published everywhere" access to information about Web services.

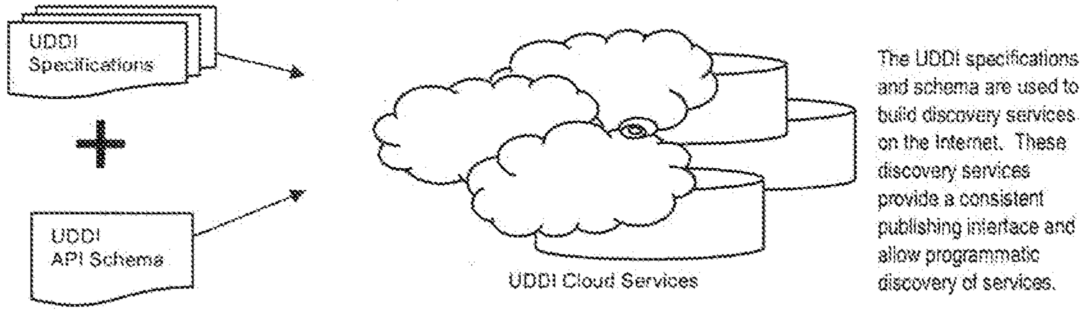

UDDI Specifications
UDDI API Schema
UDDI Cloud Services
The UDDI specifications and schema are used to build discovery services on the Internet. These discovery services provide a consistent publishing interface and allow programmatic discovery of services.

Figure 2

Using the UDDI discovery services, businesses individually register information about the Web services that they expose for use by other businesses. This information can be added to the UDDI business registry either via a Web site or by using tools that make use of the programmatic service interfaces described in the UDDI Programmer's API Specification. The UDDI business registry is a logically centralized, physically distributed service with multiple root nodes that replicate data with each other on a regular basis. Once a business registers with a single instance of the business registry service, the data is automatically shared with other UDDI root nodes and becomes freely available to anyone who needs to discover what Web services are exposed by a given business.

Next steps

As the layers in figure 1 show, it is important to note that UDDI does not form a full-featured discovery service. UDDI services are targeted at enabling technical discovery of services. With the facilities defined by UDDI, a program or programmer can locate information about services exposed by a partner, can find whether a partner has a service that is compatible with in-house technologies, and can follow links to the specifications for a Web service so that an integration layer can be constructed that will be compatible with a partners service. Businesses can also locate potential partners through UDDI directly, or more likely, from online marketplaces and search engines that use UDDI as a data source for their own value-added services. Technical compatibility can be discovered so that software companies can use the UDDI registries on the Web to automatically configure certain technical connections as software is installed or accounts are configured.

Business discovery and UDDI

UDDI is designed to complement existing online marketplaces and search engines by providing them with standardized formats for programmatic business and service discovery. The ability to locate parties that can provide a specific product or service at a given price or within a specific geographic boundary in a given timeframe is not directly covered by the UDDI specifications. These kinds of advanced discovery features require further collaboration and design work between buyer and sellers. Instead, UDDI forms the basis for defining these services in a higher layer.

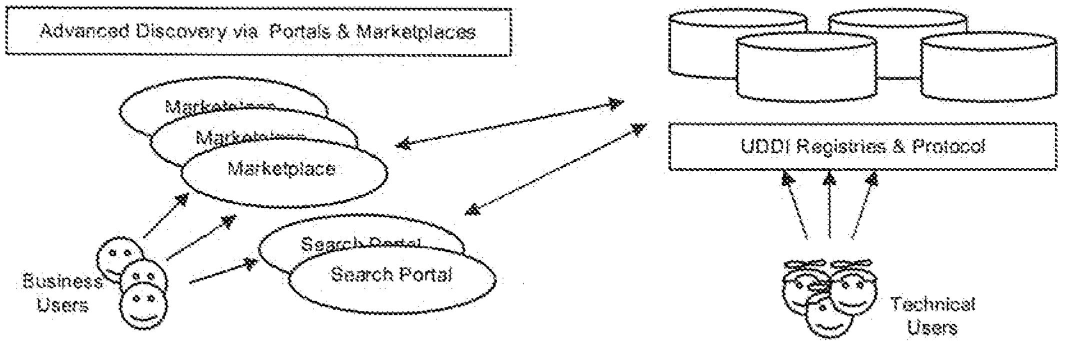


Figure 3

In Figure 3 we can see the relationship between the technical discovery layers defined by UDDI and the role of aggregation and specialized search capabilities that address business level searches. Currently, marketplaces and search portals fill this need, and can be integrated or populated using information published in the UDDI distributed registries.

Future work

The teams working on the UDDI are planning on extending the functionality beyond what is in the Open Draft specification to address more than just technical discovery. Future features will address the ability to locate products and services, define Web service implementation conventions and provide the ability to manage hierarchical business organizations, communities and trade groups. The driving goal is to provide a public specification for Web service interoperability, whether the focus is marketplace-to-marketplace or business-to-business.

The remainder of this paper is a technical overview of the various features of the UDDI discovery service and specifications.

Technical overview

The Universal Description, Discovery and Integration (UDDI) specifications consist of an XML schema for SOAP messages, and a description of the UDDI API specification. Together, these form a base information model and interaction framework that provides the ability to publish information about a broad array of Web services.

Four information types

The core information model used by the UDDI registries is defined in an XML schema. XML was chosen because it offers a platform-neutral view of data and allows hierarchical relationships to be described in a natural way. The emerging XML schema standard was chosen because of its support for rich data types as well as its ability to easily describe and validate information based on information models represented in schemas.

The UDDI XML schema defines four core types of information that provide the kinds of information that a technical person would need to know in order to use a partners Web services. These are: business information; service information, binding information; and information about specifications for services.

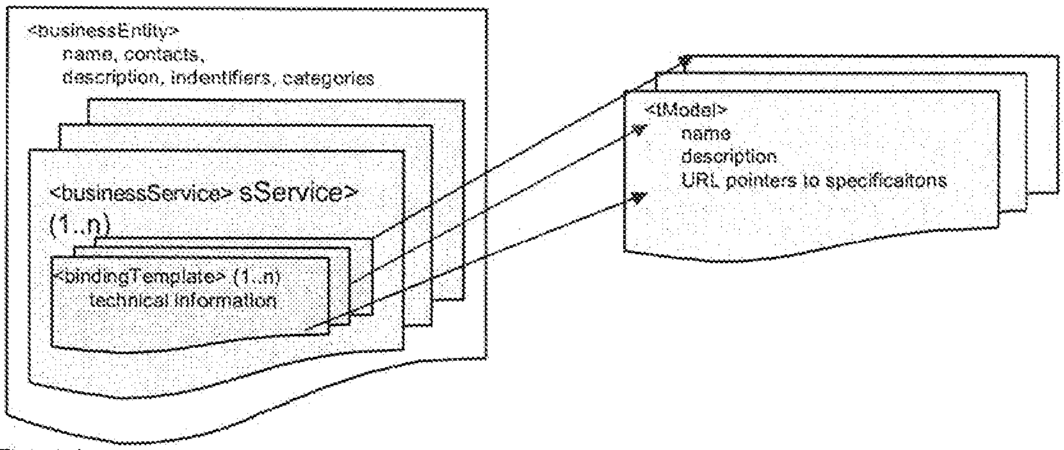


Figure 4

The information hierarchy and the key XML element names that are used to describe and discover information about Web services are shown in figure 4[1].

Business information: the businessEntity element

Many partners will need to be able to locate information about your services and will have as starting information a small set of facts about your business. Technical staff, programmers or application programs themselves will know either your business name or perhaps your business name and some key identifiers[2], as well as optional categorization and contact information. The core XML elements for supporting publishing and discovering information about a business -- the UDDI Business Registration -- are contained in a structure named "businessEntity"[3]. This structure serves as the top-level information manager for all of the information about a particular set of information related to a business unit[4].

The overall businessEntity information includes support for "yellow pages" taxonomies so that searches can be performed to locate businesses who service a particular industry or product category, or who are located within a specific geographic region.

Service information: the businessService and bindingTemplate elements

Technical and business descriptions of Web services -- the "green pages" data -- live within sub-structures of the businessEntity information. Two structures are defined: businessService and bindingTemplate. The businessService structure is a descriptive container that is used to group a series of related Web services related to either a business process or category of services. Examples of business processes that would include related Web service information include purchasing services, shipping services, and other high-level business processes.

These businessService information sets can each be further categorized -- allowing Web service descriptions to be segmented along combinations of industry, product and service or geographic category boundaries.

Within each businessService live one or more technical Web service descriptions. These contain the information that is relevant for application programs that need to connect to and then communicate with a remote Web service. This information includes the address to make contact with a Web service, as

[1] See appendix A for a more complete view of the UDDI information model.

[2] Business identifiers can include D&B numbers, tax numbers, or other information types via which partners will be able to identify a business uniquely.

[3] See the UDDI XML schema -- http://www.uddi.org/schema/uddi_1.xsd

[4] Complex business unit information should be registered in separate businessEntity records.

well as support for option information that can be used to describe both hosted services[5] and services that require additional values to be discovered prior to invoking a service[6]. Additional features are defined that allow for complex routing options such as load balancing[7].

Specification pointers and technical fingerprints

The information required to actually invoke a service is described in the information element named bindingTemplate. This was described in the previous section. However, it is not always enough to simply know where to contact a particular Web service. For instance, if I know that a business partner has a Web service that lets me send them a purchase order, knowing the URL for that service is not very useful unless I know a great deal about what format the purchase order should be sent in, what protocols are appropriate, what security required, and what form of a response will result after sending the purchase order. Integrating all parts of two systems that interact via Web services can become quite complex.

As a program or programmer interested in specific Web services, information about compatibility with a given specification is required to make sure that the right Web service is invoked for a particular need. For this reason, each bindingTemplate element contains a special element that is a list of references to information about specifications. Used as an opaque[8] set of identifiers, these references form a technical fingerprint that can be used to recognize a Web service that implements a particular behavior or programming interface.

In our purchase order example, the Web service that accepts the purchase order exhibits a set of well-defined behaviors if the proper document format is sent to the proper address in the right way. A UDDI registration for this service would consist of an entry for the business partner, a logical service entry that describes the purchasing service, and a bindingTemplate entry that describes the purchase order service by listing its URL and a reference to a tModel.

These references are actually the keys that can be used to access information about a specification. Called "tModels", this information is *metadata* about a specification, including its name, publishing organization, and URL pointers[9] to the actual specifications themselves. In our example, the tModel reference found in the bindingTemplate is a pointer to information about the specifics of this purchase order Web service. The reference itself is a pledge by the company that exposes the Web service that they have implemented a service that is compatible with the tModel that is referenced. In this way, many companies can provide Web services that are compatible with the same specifications.

The programmer's API

The UDDI specifications include definitions for Web service interfaces that allow programmatic access to the UDDI registry information. The full definition of the programmer's API is found in the Programmer's API Specification document. The API capabilities are briefly discussed below.

The API is divided into two logical parts. These are the Inquiry API and the Publishers' API. The Inquiry API is further divisible into two parts – one part used for constructing programs that let you search and browse information found in a UDDI registry, and another part that is useful in the event that Web service invocations experience failures. Programmers can use the Publishers API to create rich

[5] Other companies provision hosted services, typically on a fee base. Marketplaces are good examples of hosted services.

[6] Software packages are a good example of this kind of requirement. Individual installations of a given package may have specific values that must be accommodated prior to connecting with a service.

[7] The hostingRedirector feature allows both hosting and other redirection capabilities to be deployed. See the appendix on redirection in the UDDI programmers API specification for more details.

[8] The term opaque in this context alludes to the fact that simply knowing a key value for a particular specification is equivalent to knowing a service is compatible with that specification.

[9] **Note**: tModels do not actually contain the actual specifications. UDDI defines a framework for taking advantage of URLs and web servers, so that individual organizations can maintain centralized specifications. Individual implementations can then be located based on whether or not they contain references to specific specification keys.

interfaces for tools that interact directly with a UDDI registry, letting a technical person manage the information published about either a businessEntity or a tModel structure.

Built on SOAP

The Simple Object Access Protocol (SOAP) is a W3C draft note describing a way to use XML and HTTP to create an information delivery and remote procedure mechanisms. Several companies, including IBM, Microsoft, DevelopMentor and Userland Software, submitted this draft note to the W3C for the purpose of, among other things, standardizing RPC (simple messaging) conventions on the World Wide Web. In its current state, the draft note describes a specification that is useful for describing a Web service. The companies that collaborated on UDDI decided to base the UDDI APIs on this SOAP specification. The specifics of how SOAP and XML are used by UDDI registry *Operators* are defined in the appendices in the API specification itself.

All of the API calls defined by the UDDI Programmer's API Specification behave synchronously – and all of the distributed UDDI registry *Operator Sites* support all of the calls described in the Programmer's API Specification.

The Inquiry API

The Inquiry API consists of two types of calls that let a program quickly locate candidate businesses, Web services and specifications, and then drill into specifics based on overview information provided in initial calls. The APIs named *find_xx* provide the caller with a broad overview of registration data based on a variety of search criteria. Alternately, if the actual keys of specific data are known ahead of time, up to date copies of a particular structure (e.g. businessEntity, businessService, bindingTemplate, tModel) can be retrieved in full via a direct call. These direct calls are called the *get_xx* APIs.

The UDDI invocation model

Each individual advertised Web service is modeled in a bindingTemplate structure. Invocation of a Web service is typically performed based on cached bindingTemplate data. With this in mind, the general scenario for using UDDI becomes clear when you consider the preparation required to write a program that uses a specific Web service. The following recipe outlines these steps

1. The programmer, chartered to write a program that uses a remote Web service, uses the UDDI business registry (either via a Web interface or other tool that uses the Inquiry API) to locate the businessEntity information registered by or for the appropriate business partner that is advertising the Web service.
2. The programmer either drills down for more detail about a businessService or requests a full businessEntity structure. Since businessEntity structures contain all information about advertised Web services, the programmer selects a particular bindingTemplate[10] and saves this away for later use.
3. The programmer prepares the program based on the knowledge of the specifications for the Web service. This information may be obtained by using the tModel key information contained in the bindingTemplate for a service.
4. At runtime, the program invokes the Web service as planned using the cached bindingTemplate information (as appropriate).

In the general case, assuming the remote Web service and the calling program each accurately implement the required interface conventions (as defined in the specification referenced in the tModel information), the calls to the remote service will function successfully. The special case of failures and recovery is outlined next.

[10] Using the find_xx inquiry API, a UDDI compatible browser can display more or less detail as someone searches through information. Once the appropriate information is located, the get_xx call returns full information about one of the four key UDDI XML structures.

Recovery after remote Web service call failure

One of the key benefits of maintaining information about Web services in a distributed UDDI Registry is the "self service" capability provided to technical personnel. The recipe in the previous section outlined the tasks that the programmer is able to accomplish using the information found in the UDDI registry. This is all fine and well, but additional benefits are possible. These benefits of using a distributed UDDI registry with information hosted at an *Operator Site* are manifested in disaster recovery scenarios.

Web services businesses using Web services to do commerce with their partners need to be able to detect and manage communication problems or other failures. A key concern is the inability to predict, detect, or recover from failures within the systems of the remote partner. Even simple situations such as temporary outages caused by nightly maintenance or back-ups can make the decision to migrate to Web services difficult.

On the other hand, if you are the company that makes direct Web service connections possible, disaster recovery and the ability to migrate all of your business partners to a back-up system are prime concerns.

UDDI starts to address these "quality of service" issues by defining a calling convention that involves using cached bindingTemplate information, and when failures occur, refreshing the cached information with current information from a UDDI Web registry. The recipe for this convention goes like this:

1. Prepare program for Web service, caching the required bindingTemplate data for use at run-time.
2. When calling the remote Web service, use the cached bindingTemplate data that was obtained from a UDDI Web registry.
3. If the call fails, use the bindingKey value and the get_bindingTemplate API call to get a fresh copy of a bindingTemplate for this unique Web service.
4. Compare the new information with the old – if it is different, retry the failed call. If the retry succeeds, replace the cached data with the new data.

Behind the scenes, when a business needs to redirect traffic to a new location or backup system, they only need to activate the backup system and then change the published location information for the effected bindingTemplates. This approach is called *retry on failure* – and is more efficient than getting a fresh copy of bindingTemplate data prior to each call.

The Publication API

The Publication API consists of four *save_xx* functions and four delete_xx functions, one each for the four key UDDI data structures (businessEntity, businessService, bindingTemplate, tModel). Once authorized, an individual party can register any number of businessEntity or tModel information sets, and can alter information previously published. The API design model is simple – changes to specific related information can be made and new information be saved using save. Complete structure deletion is accommodated by the delete calls. See the Programmer's API Specification for more information on this topic.

Security: Identity and authorization

The key operating principal for the UDDI Publishers' API is to only allow authorized individuals to publish or change information within the UDDI business registry. Each of the individual implementations of the distributed UDDI business registry maintains a unique list of authorized parties and tracks which businessEntity or tModel data was created by a particular individual. Changes and deletions are only allowed if a change request (via API call) is made by the same individual who created the effected information.

Each instance of a UDDI business registry, called an *Operator Site*, is allowed to define its own end user authentication mechanism[11], but all of the contracted UDDI *Operator Sites* are required to meet certain minimum criteria that provide similar security protections.

Other information

For more details on the UDDI schema or the UDDI Programmer's API Specification, consult the documents that are available on the uddi.org Web site.

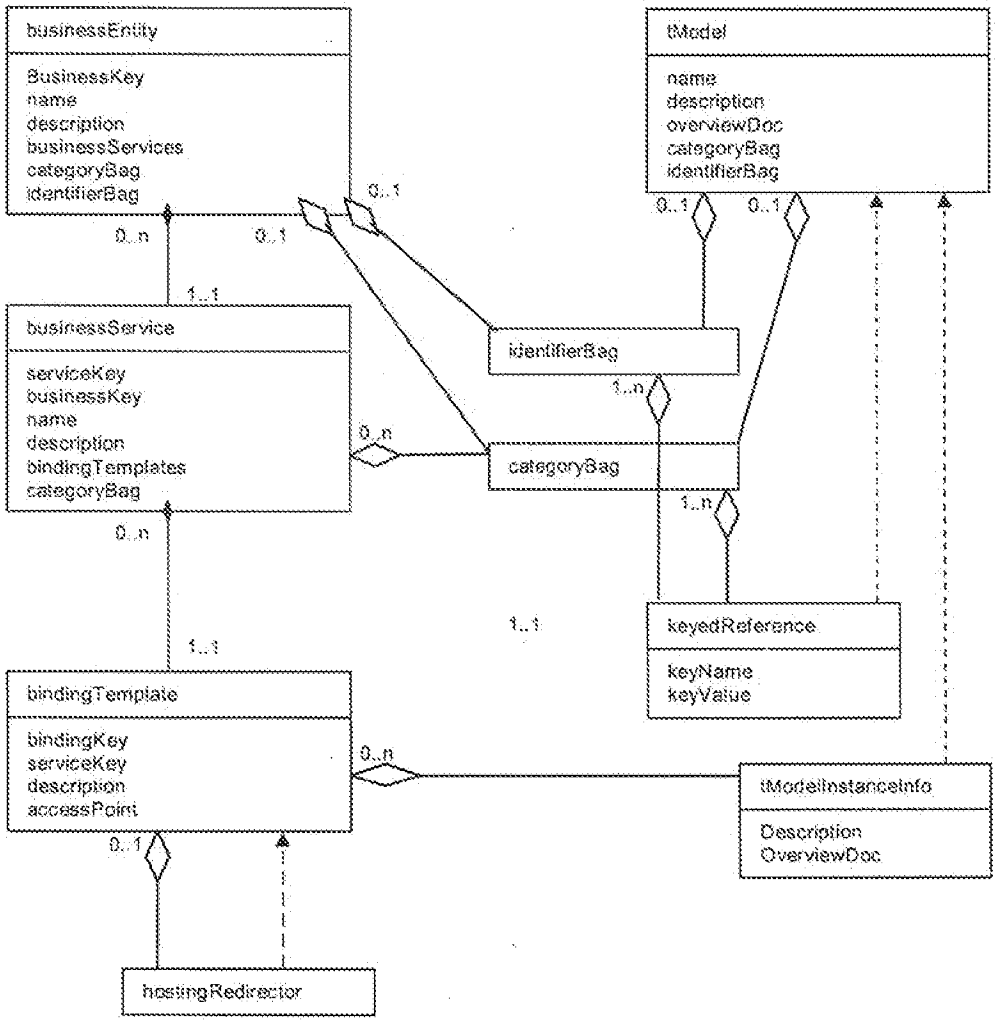
These specifications are published as a set of linked HTML documents, with specific tModels defined for related sub service offerings. The individual tModel references are actually overview information with shared links to overview, API call, and appendix information within the overall specification.

Printable versions of the full specification in Microsoft Word and PDF format will also be available for download.

[11] Private implementations that are built using the UDDI specifications cannot be forced to implement any specific conventions or requirements. For this reason, be sure to consult with the UDDI implementer if you have questions about security or information access control policies.

Appendix A: UDDI information model

The diagram below shows the relationships between the different core information elements that make up the UDDI information model.



Resources

This section contains the locations of various specifications, document references and useful information where you can learn more about this subject.

- W3C XML and related recommendations: http://www.w3.org/TR
- SOAP 1.1 W3C note: http://www.w3.org/TR/#Notes
- UDDI Web site: http://www.uddi.org
    - UDDI Programmer's API Specification: http://www.uddi.org/pubs/UDDI_Programmers_API_Specification.pdf
    - UDDI Data Structure Reference: http://www.uddi.org/pubs/UDDI_XML_Structure_Reference.pdf
    - UDDI Revision 1.0 schema: http://www.uddi.org/schema/uddi_1.xsd

Modeling the UDDI Schema with UML

Exhibit B

Dave Carlson
CTO
Ontogenics Corp.
Boulder, Colorado
dcarlson@ontogenics.com
http://XMLModeling.com

Complex XML vocabulary definitions are often easier to comprehend and discuss with others when they are expressed graphically. Although existing tools for editing schemas provide some assistance in this regard (e.g., Extensibility XML Authority) they are generally limited to a strict hierarchical view of the vocabulary structure. More complex structures are often represented in schemas using a combination of containment and link references (including ID/IDREF, simple href attributes, and more flexible extended XLink attributes). These more object-oriented models of schema definition are more easily represented using UML class diagrams [1].

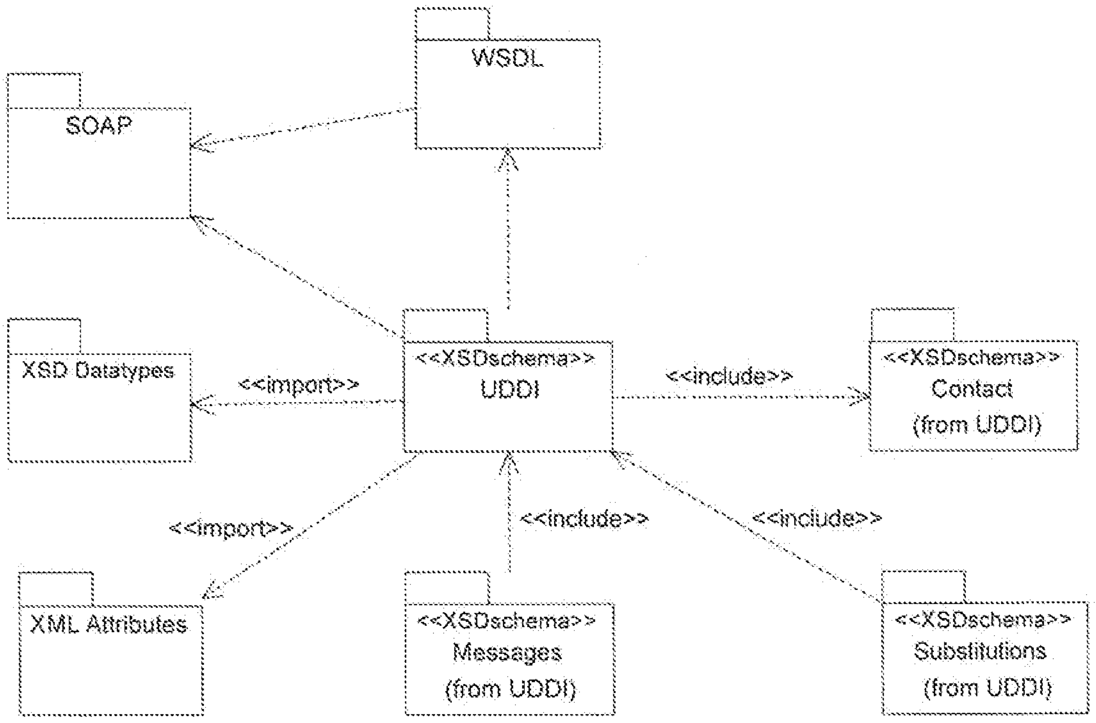
The Unified Modeling Language (UML) is the most recent evolution of graphical notations for object-oriented analysis and design, plus it includes the necessary metamodel that defines the modeling language itself. UML has experienced rapid growth and adoption in the last two years, due in part to its vendor-independent specification as a standard by the Object Management Group (OMG). This UML standard and metamodel are the basis for a new wave of modeling tools by many vendors. For more information and references to these topics, see the Web portal at XMLModeling.com, which was created for the purpose of communicating information about the intersection of XML and UML technologies.

The OMG has also adopted a standard interchange format for serializing models and exchanging them between UML tools, called the XML Metadata Interchange (XMI) specification [2]. (XMI is actually broader in scope than this, but its use with UML is most prevalent.) Many UML modeling tools now support import/export using the XMI format, so it's now possible to get an XML document that contains a complete UML model definition. This capability is the foundation for the remainder of this white paper. Using an XMI file exported from any UML tool, I have written an XSLT transformation that generates an XML Schema document from the UML class model definition.

As an example that demonstrates the benefits of modeling XML vocabularies with UML, I have reverse-engineered a substantial part of the UDDI specification [3]. The current UDDI specification (as of January 2001) includes an XML Schema definition that is based on an old version of the XML Schema draft, well before the out-dated April 7th 2000 draft. In contrast, the UDDI schemas described here are compliant with the XML Schema Candidate Recommendation, dated October 24, 2000. The reverse engineering was accomplished manually, by reading the UDDI specification and creating a UML model in Rational Rose.

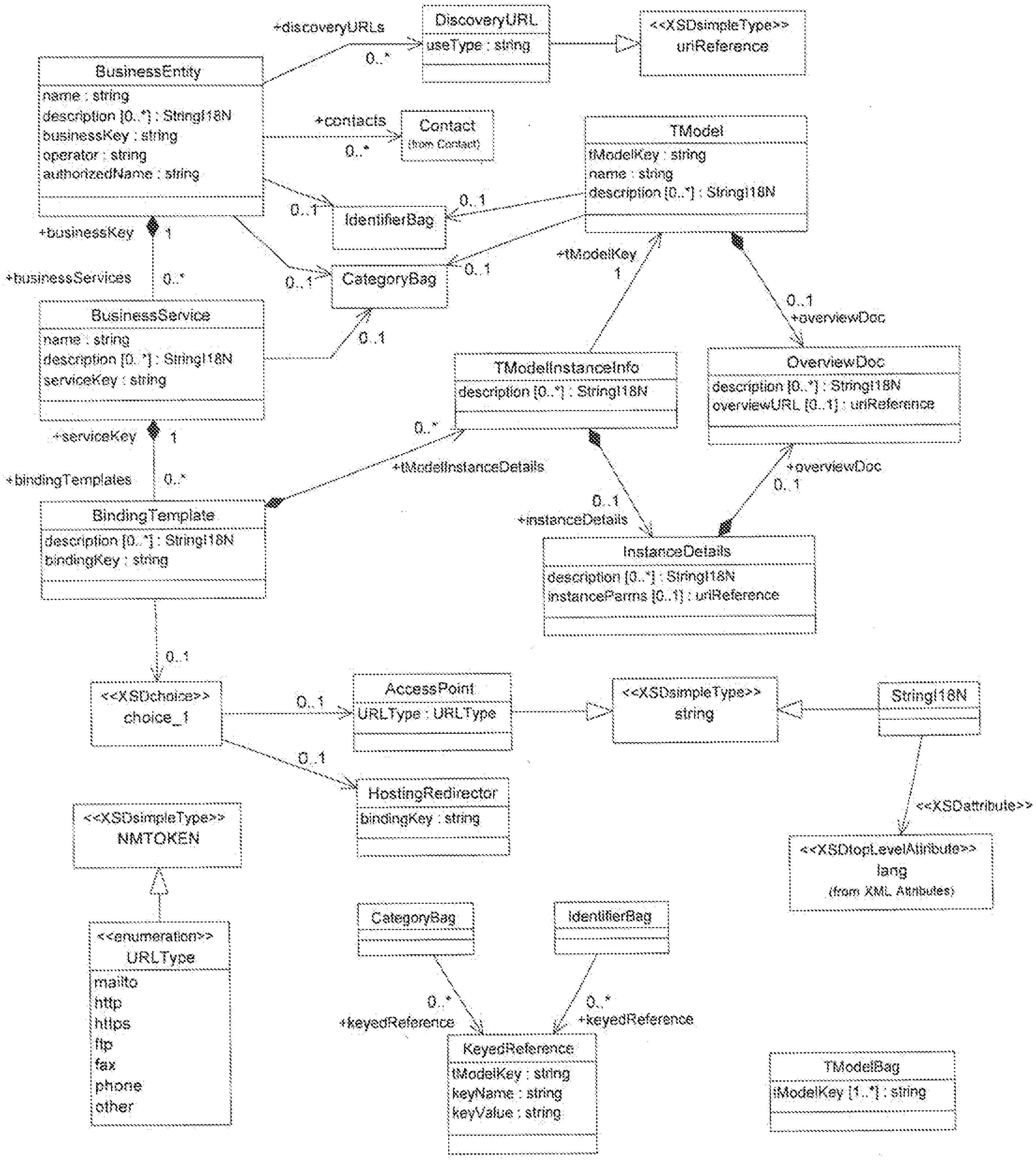
This is work in progress! I spent only a few days studying the UDDI specification and schema while building this model. I then applied a prototype tool that generates XML Schemas from UML, with the goal of regenerating a schema that is semantically equivalent to the one included in the UDDI specification. I used the jUDDI client application [4] to query a business description document from Microsoft's UDDI repository. That XML document is, of course, represented using elements from the UDDI schema. Finally, using the XML Spy 3.5 beta tool, the UDDI instance document was successfully validated using the schema generated from this UML model.

UML enables definition of a model structure using packages; these are illustrated in a diagram using file folder icons, optionally connected with dependency arrows. A high-level view of UDDI and its relationship to other specifications is shown in the following figure. The following examples expand the contents of the three packages named UDDI, Contact, and Messages in the diagram.



UDDI Core Content Model

The following UML class diagram illustrates all of the core elements from the UDDI specification. There are several important points to notice. First, I followed a common software design practice when naming the classes by beginning class names with an upper-case letter; the UDDI schema uses lower-case letters to begin all element names. Several other prominent XML schemas also adopt the more object-oriented naming conventions as used here. This design requires additional UML and schema elements to be declared that map lower-case names to their upper-case complexType definitions. Second, in order to allow a designer control over the XML schema generation process, the model was defined with several UML stereotypes and tagged values (a standard technique that is part of UML). Most of these are not shown in this diagram, but they did affect the generated schema. Although you can generate a schema from this UML model without using any stereotype extensions, it is unlikely to match the structure of a reverse-engineered schema without some fine-tuning.


BusinessEntity
name : string
description [0..*] : StringI18N
businessKey : string
operator : string
authorizedName : string
+discoveryURLs
0..*
DiscoveryURL
useType : string
<<XSDsimpleType>>
uriReference
+contacts
0..*
Contact
(from Contact)
TModel
tModelKey : string
name : string
description [0..*] : StringI18N
0..1
IdentifierBag
0..1
+businessKey
1
+businessServices
0..*
0..1
CategoryBag
0..1
+tModelKey
1
BusinessService
name : string
description [0..*] : StringI18N
serviceKey : string
0..1
0..1
+overviewDoc
TModelInstanceInfo
description [0..*] : StringI18N
OverviewDoc
description [0..*] : StringI18N
overviewURL [0..1] : uriReference
+serviceKey
1
+bindingTemplates
0..*
0..*
+tModelInstanceDetails
+overviewDoc
0..1
0..1
+instanceDetails
BindingTemplate
description [0..*] : StringI18N
bindingKey : string
InstanceDetails
description [0..*] : StringI18N
instanceParms [0..1] : uriReference
0..1
<<XSDchoice>>
choice_1
0..1
AccessPoint
URLType : URLType
<<XSDsimpleType>>
string
StringI18N
0..1
HostingRedirector
bindingKey : string
<<XSDattribute>>
<<XSDsimpleType>>
NMTOKEN
<<XSDtopLevelAttribute>>
lang
(from XML Attributes)
<<enumeration>>
URLType
mailto
http
https
ftp
fax
phone
other
CategoryBag
IdentifierBag
0..*
+keyedReference
0..*
+keyedReference
KeyedReference
tModelKey : string
keyName : string
keyValue : string
TModelBag
tModelKey [1..*] : string

I chose the UDDI schema as the subject of this reverse-engineering exercise in part because of its complexity. If this can be accomplished successfully, then other projects are also likely to be feasible. In addition, if the UML modeling was done as an integral part of the original schema analysis and design, then a number of inconsistencies could be resolved and the schema could be generated using fewer stereotype extensions.

XML Schema definitions are shown for four of the UML model classes. These were generated from the XMI export of this model using an XSLT stylesheet to execute the transformations. They were not manually edited. The schema generator also includes full support for generalization (i.e., inheritance) in the UML model and produces valid XML Schema definitions using both complexType and simpleType derivation. The UML stereotypes and associated tagged values provide control over design issues such as: selection of <choice>, <sequence>, or <all> content model for each complexType definition; specification of the element order within <sequence> (UML attributes are unordered in the model); selection of whether UML attributes and association roles are generated as attributes or elements in the XML Schema; and control over the way that UML association ends are embedded with or without role elements, and as 'ref' or 'type' declarations.

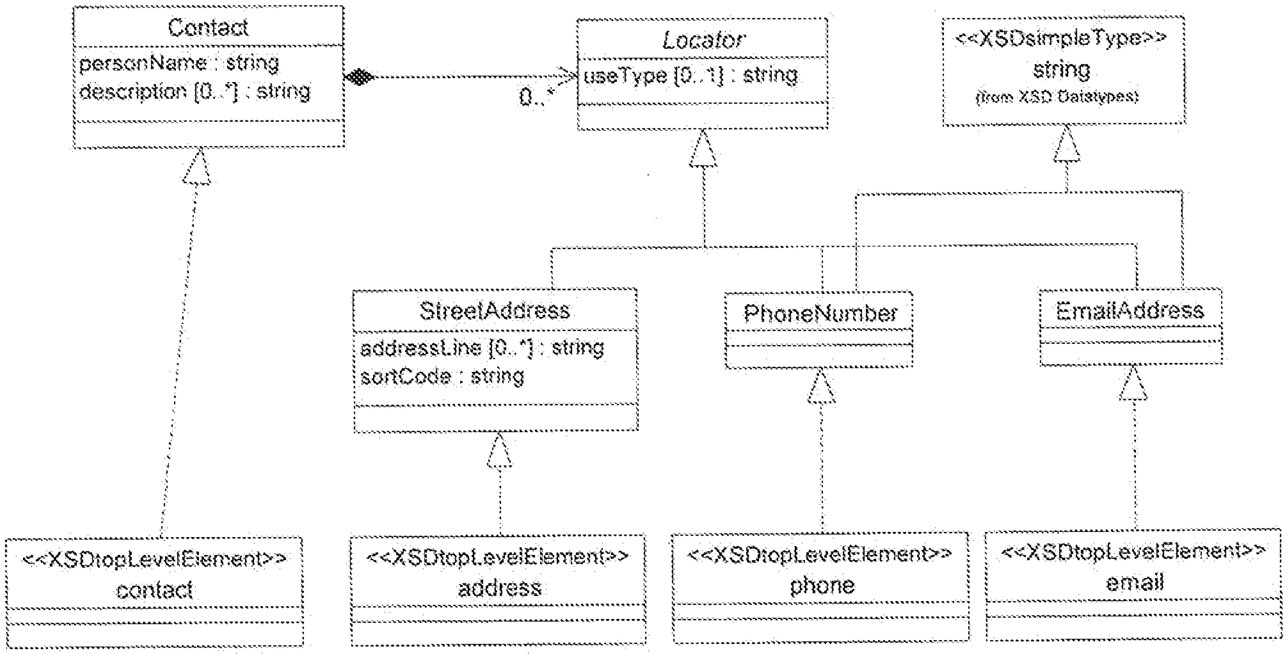
```
<!-- ============================================================= -->
<!-- CLASS: BusinessEntity  -->
<!-- ============================================================= -->

  <xs:element name="BusinessEntity" type="uddi:BusinessEntity"/>
  <xs:complexType name="BusinessEntity">
     <xs:annotation>
        <xs:documentation>
          Describes an instance of a business or business unit.
        </xs:documentation>
     </xs:annotation>
     <xs:sequence>
        <xs:element name="discoveryURLs" minOccurs="0" maxOccurs="1">
           <xs:complexType>
              <xs:sequence>
                 <xs:element ref="uddi:DiscoveryURL"
                             minOccurs="0" maxOccurs="unbounded"/>
              </xs:sequence>
           </xs:complexType>
        </xs:element>
        <xs:element name="name" type="xsd:string"/>
        <xs:element name="description" type="uddi:StringI18N"
                    minOccurs="0" maxOccurs="unbounded"/>
        <xs:element name="contacts" minOccurs="0" maxOccurs="1">
           <xs:complexType>
              <xs:sequence>
                 <xs:element ref="uddi:Contact"
                             minOccurs="0" maxOccurs="unbounded"/>
              </xs:sequence>
           </xs:complexType>
        </xs:element>
        <xs:element name="businessServices" minOccurs="0" maxOccurs="1">
           <xs:complexType>
              <xs:sequence>
                 <xs:element ref="uddi:BusinessService"
```

```
                              minOccurs="0" maxOccurs="unbounded"/>
                  </xs:sequence>
               </xs:complexType>
            </xs:element>
            <xs:element ref="uddi:IdentifierBag" minOccurs="0" maxOccurs="1"/>
            <xs:element ref="uddi:CategoryBag" minOccurs="0" mazOccurs="1"/>
         </xs:sequence>
         <xs:attributeGroup ref="uddi:BusinessEntity.att"/>
      </xs:complexType>
      <xs:attributeGroup name="BusinessEntity.att">
         <xs:attribute name="businessKey" type="xsd:string" use="required"/>
         <xs:attribute name="operator" type="xsd:string" use="required"/>
         <xs:attribute name="authorizedName" type="xsd:string" use="required"/>
      </xs:attributeGroup>

   <!-- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ -->
   <!-- CLASS: BusinessService  -->
   <!-- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ -->

      <xs:element name="BusinessService" type="uddi:BusinessService"/>

      <xs:complexType name="BusinessService">
         <xs:annotation>
            <xs:documentation>Describes a logical service type in business terms.
            </xs:documentation>
         </xs:annotation>
         <xs:sequence>
            <xs:element name="name" type="xsd:string"/>
            <xs:element name="description" type="uddi:StringI18N"
                        minOccurs="0" maxOccurs="unbounded"/>
            <xs:element name="bindingTemplates" minOccurs="0" maxOccurs="1">
               <xs:complexType>
                  <xs:sequence>
                     <xs:element ref="uddi:BindingTemplate"
                              minOccurs="0" maxOccurs="unbounded"/>
                  </xs:sequence>
               </xs:complexType>
            </xs:element>
            <xs:element ref="uddi:CategoryBag" minOccurs="0" mazOccurs="1"/>
         </xs:sequence>
         <xs:attributeGroup ref="uddi:BusinessService.att"/>
      </xs:complexType>
      <xs:attributeGroup name="BusinessService.att">
         <xs:attribute name="serviceKey" type="xsd:string" use="required"/>
         <xs:attribute name="businessKey" type="xs:string"/>
      </xs:attributeGroup>

   <!-- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ -->
   <!-- CLASS: CategoryBag  -->
   <!-- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ -->

      <xs:element name="CategoryBag" type="uddi:CategoryBag"/>
      <xs:complexType name="CategoryBag">
         <xs:sequence>
            <xs:element name="keyedReference" type="uddi:KeyedReference"
                        minOccurs="0" maxOccurs="unbounded"/>
```

```
      </xs:sequence>
      <xs:attributeGroup ref="uddi:CategoryBag.att"/>
   </xs:complexType>
   <xs:attributeGroup name="CategoryBag.att"/>

<!-- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ -->
<!-- CLASS: KeyedReference  -->
<!-- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ -->

   <xs:element name="KeyedReference" type="uddi:KeyedReference"/>
   <xs:complexType name="KeyedReference">
      <xs:attributeGroup ref="uddi:KeyedReference.att"/>
   </xs:complexType>
   <xs:attributeGroup name="KeyedReference.att">
      <xs:attribute name="tModelKey" type="xsd:string" use="required"/>
      <xs:attribute name="keyName" type="xsd:string" use="required"/>
      <xs:attribute name="keyValue" type="xsd:string" use="required"/>
   </xs:attributeGroup>
```

Contact Package

These elements related to business contact information are part of the core UDDI schema. However, I have created a separate package in this UML model for three reasons. First, it's important to promote the separation and reuse of common components. This is a fundamental goal in object-oriented analysis and design, and contact or party information is a classic example of common elements required in most e-business applications. Second, because this is primarily an educational exercise, I wanted to create a second UML package that would be included by the schema transformation tool. Third, and also part of the educational goal, this package includes an experimental mapping for multiple inheritance in UML models to XML Schema. Schema does not support multiple inheritance.



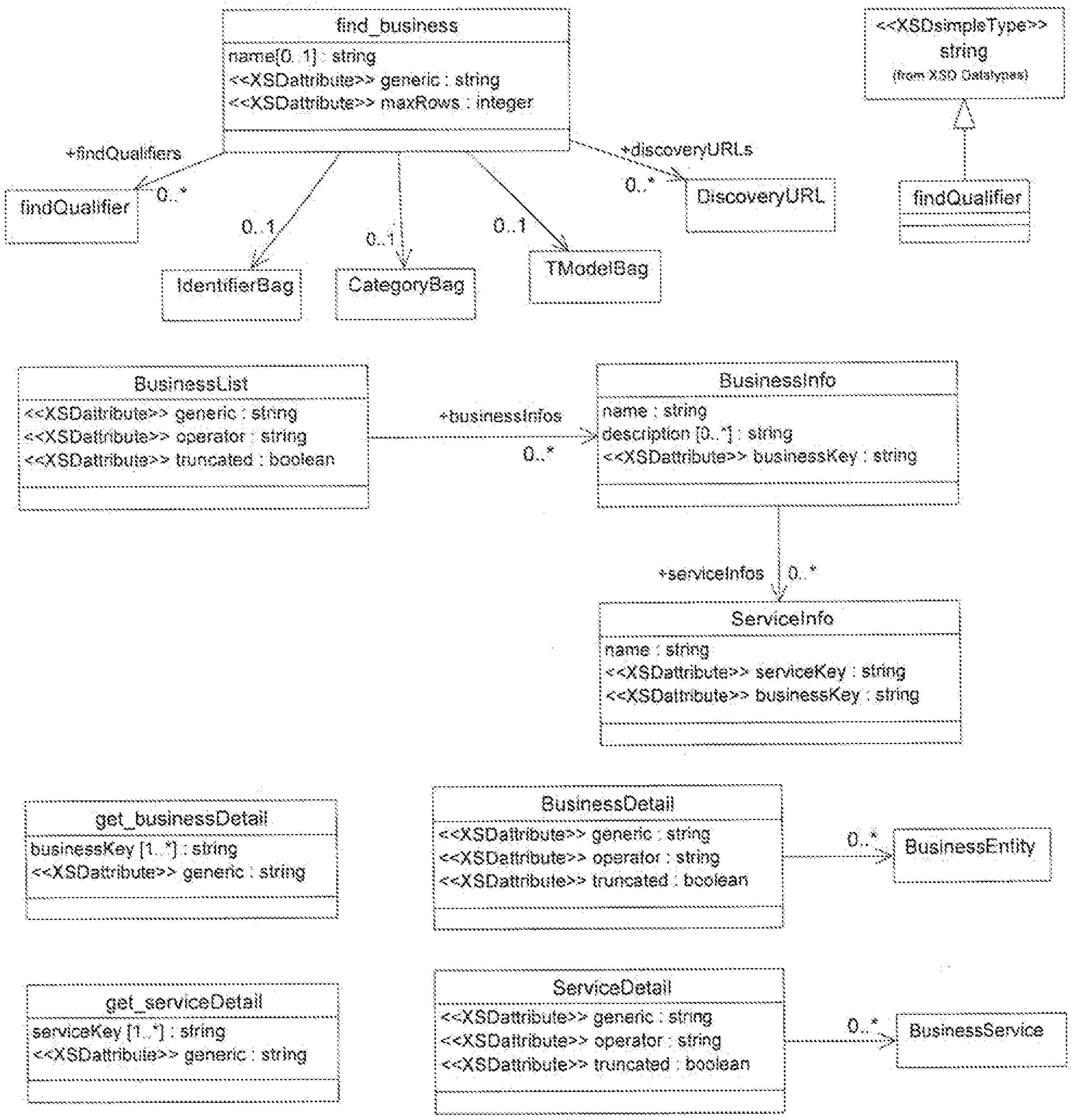
The Locator class is an abstract class in UML (denoted by its name in italics in the diagram), so it is generated to a complexType with an abstract='true' attribute. StreetAddress uses single inheritance, so it may use complexType extension to inherit the attributes of Locator in the schema. PhoneNumber inherits from both Locator (a complexType) and string (a simpleType drawn from the XSD Datatypes package). In this mapping, a simpleType superclass takes precedence and generates the corresponding simpleContent child in PhoneNumber. The Locator attributes are copied down and duplicated within the content model for PhoneNumber. Finally, all three subclasses include a substitutionGroup attribute that allows them to be substituted for Locator within the content of a Contact element.

As mentioned previously, this model uses upper-case names for the model elements, whereas the UDDI schema uses lower-case names. Thus, this model includes four additional classes stereotyped as << XSDtopLevelElement>> that create lower-case elements in the generated schema (not shown in the code examples). These classes are connected with a UML "realization" relationship in this mapping. This contact model creates a schema that is slightly more lenient than the original UDDI schema, but it successfully validates current UDDI documents and I believe that it more accurately reflects the intended conceptual model of UDDI requirements (with the exception of upper/lower-case issues).

```
<!-- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ -->
<!-- CLASS: Locator  -->
<!-- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ -->

   <xs:element name="Locator" type="uddi:Locator"/>
   <xs:complexType name="Locator" abstract="true">
      <xs:attributeGroup ref="uddi:Locator.att"/>
   </xs:complexType>
   <xs:attributeGroup name="Locator.att">
      <xs:attribute name="useType" type="xsd:string"/>
   </xs:attributeGroup>

<!-- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ -->
<!-- CLASS: StreetAddress  -->
<!-- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ -->

   <xs:element name="StreetAddress" type="uddi:StreetAddress"
               substitutionGroup="uddi:Locator"/>
   <xs:complexType name="StreetAddress">
      <xs:annotation>
         <xs:documentation>
           A printable, free-form address. Typed by convention. Sort not used.
         </xs:documentation>
      </xs:annotation>
      <xs:complexContent>
         <xs:extension base="uddi:Locator">
            <xs:sequence>
               <xs:element name="addressLine" type="xsd:string"
                           minOccurs="0" maxOccurs="unbounded"/>
            </xs:sequence>
            <xs:attributeGroup ref="uddi:StreetAddress.att"/>
         </xs:extension>
      </xs:complexContent>
   </xs:complexType>
   <xs:attributeGroup name="StreetAddress.att">
      <xs:attribute name="sortCode" type="xsd:string" use="required"/>
   </xs:attributeGroup>

<!-- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ -->
<!-- CLASS: PhoneNumber  -->
<!-- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ -->

   <xs:element name="PhoneNumber" type="uddi:PhoneNumber"
               substitutionGroup="uddi:Locator"/>
   <xs:complexType name="PhoneNumber">
      <xs:simpleContent>
         <xs:extension base="xsd:string">
            <xs:attributeGroup ref="uddi:Locator.att"/>
            <xs:attributeGroup ref="uddi:PhoneNumber.att"/>
         </xs:extension>
      </xs:simpleContent>
   </xs:complexType>
   <xs:attributeGroup name="PhoneNumber.att"/>
```

UDDI Message Model

The UDDI specification is defined by a single schema and that schema also includes elements that are the basis for SOAP messages. I split these definitions into two packages in the UML model and both are generated into a single schema document. The following diagram shows a few classes that represent a small subset of the UDDI message structures.


find_business
name[0..1] : string
<<XSDattribute>> generic : string
<<XSDattribute>> maxRows : integer
+findQualifiers
0..*
findQualifier
0..1
IdentifierBag
0..1
CategoryBag
0..1
TModelBag
+discoveryURLs
0..*
DiscoveryURL
<<XSDsimpleType>>
string
(from XSD Datatypes)
findQualifier
BusinessList
<<XSDattribute>> generic : string
<<XSDattribute>> operator : string
<<XSDattribute>> truncated : boolean
+businessInfos
0..*
BusinessInfo
name : string
description [0..*] : string
<<XSDattribute>> businessKey : string
+serviceInfos
0..*
ServiceInfo
name : string
<<XSDattribute>> serviceKey : string
<<XSDattribute>> businessKey : string
get_businessDetail
businessKey [1..*] : string
<<XSDattribute>> generic : string
BusinessDetail
<<XSDattribute>> generic : string
<<XSDattribute>> operator : string
<<XSDattribute>> truncated : boolean
0..*
BusinessEntity
get_serviceDetail
serviceKey [1..*] : string
<<XSDattribute>> generic : string
ServiceDetail
<<XSDattribute>> generic : string
<<XSDattribute>> operator : string
<<XSDattribute>> truncated : boolean
0..*
BusinessService

The remaining examples focus on two of these UDDI messages: `get_businessDetail` and `BusinessDetail`. These messages represent a request—response pair in the UDDI protocol and both are wrapped in SOAP message envelopes when exchanged by applications. Notice that the `BusinessDetail` includes an association to zero or more `BusinessEntity` elements, which reference the definition listed in the previous section. Notice also that an extra top-level element was declared for `businessDetail` (not shown in the diagram) that maps to the upper-case complexType named `BusinessDetail` in the schema.

```
<!-- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ -->
<!-- CLASS: get_businessDetail  -->
<!-- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ -->

   <xs:element name="get_businessDetail" type="msg:get_businessDetail"/>
   <xs:complexType name="get_businessDetail">
      <xs:sequence>
         <xs:element name="businessKey" type="xsd:string"
                     minOccurs="1" maxOccurs="unbounded"/>
      </xs:sequence>
      <xs:attributeGroup ref="msg:get_businessDetail.att"/>
   </xs:complexType>
   <xs:attributeGroup name="get_businessDetail.att">
      <xs:attribute name="generic" type="xsd:string" use="required"/>
   </xs:attributeGroup>

<!-- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ -->
<!-- CLASS: BusinessDetail  -->
<!-- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ -->

   <xs:element name="BusinessDetail" type="msg:BusinessDetail"/>
   <xs:complexType name="BusinessDetail">
      <xs:sequence>
         <xs:element ref="uddi:BusinessEntity"
                     minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
      <xs:attributeGroup ref="msg:BusinessDetail.att"/>
   </xs:complexType>
   <xs:attributeGroup name="BusinessDetail.att">
      <xs:attribute name="generic" type="xsd:string" use="required"/>
      <xs:attribute name="operator" type="xsd:string" use="required"/>
      <xs:attribute name="truncated" type="xsd:boolean" use="required"/>
   </xs:attributeGroup>

<!-- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ -->
<!-- CLASS: <<XSDtopLevelElement>> businessDetail  -->
<!-- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ -->

   <xs:element name="businessDetail" type="msg:BusinessDetail"
               substitutionGroup="msg:BusinessDetail"/>
```

Sample UDDI <businessDetail> Document

When a `<get_businessDetail>` message is sent to Microsoft's UDDI repository (wrapped in a SOAP message envelope), then the following `<businessDetail>` message is returned. I've removed the SOAP envelope elements that wrapped this response message and modified the default namespace declaration to assign my test namespace, but otherwise the message is exactly as returned.

```
<businessDetail generic="1.0" operator="Microsoft Corporation" truncated="false"
    xmlns="http://www.xmlmodeling.com/schemas/uddi"
    xmlns:xsi="http://www.w3.org/2000/10/XMLSchema-instance"
    xsi:schemaLocation="http://www.xmlmodeling.com/schemas/uddi
                        uddi.xsd">
    <!-- Dave Carlson:
         Modified original default namespace from xmlns="urn:uddi-org:api"
    -->

  <businessEntity authorizedName="Martin Kohlleppel"
                  businessKey="0076B468-EB27-42E5-AC09-9955CFF462A3"
                  operator="Microsoft Corporation">
    <discoveryURLs>
      <discoveryURL useType="businessEntity">
          http://uddi.microsoft.com/discovery?businessKey=0076B468-EB27-42E5-AC09-
9955CFF462A3</discoveryURL>
    </discoveryURLs>
    <name>Microsoft Corporation</name>
    <description xml:lang="en">Software Vendor</description>
    <contacts>
      <contact>
        <personName>Martin Kohlleppel</personName>
        <email>martink@microsoft.com</email>
      </contact>
    </contacts>
    <businessServices>
      <businessService businessKey="0076B468-EB27-42E5-AC09-9955CFF462A3"
                       serviceKey="CC02E21A-2B4E-4F1A-A2C0-5C5952D67231">
        <name>UDDI Web Services</name>
        <description xml:lang="en">UDDI Registries and Test Sites</description>
        <bindingTemplates>
          <bindingTemplate bindingKey="2303E786-68C0-426E-A646-8E366B43F1FD"
                           serviceKey="CC02E21A-2B4E-4F1A-A2C0-5C5952D67231">
            <description xml:lang="en">
                Microsoft Production UDDI Registry web interface
            </description>
            <accessPoint URLType="http">http://uddi.microsoft.com</accessPoint>
            <tModelInstanceDetails>
              <tModelInstanceInfo
                    tModelKey="uuid:4CD7E4BC-648B-426D-9936-443EAAC8AE23">
                <description xml:lang="en">UDDI Inquiry Service Type</description>
                <instanceDetails>
                  <description xml:lang="en">SOAP URL for inquire</description>
                  <instanceParms>http://uddi.microsoft.com/inquire</instanceParms>
                </instanceDetails>
              </tModelInstanceInfo>
              <tModelInstanceInfo
                    tModelKey="uuid:64C756D1-3374-4E00-AE83-EE12E38FAE63">
                <description xml:lang="en">UDDI Inquiry Publication Type
                </description>
                <instanceDetails>
                  <description xml:lang="en">SOAP URL for publication</description>
                  <instanceParms>https://uddi.microsoft.com/publish</instanceParms>
                </instanceDetails>
              </tModelInstanceInfo>
            </tModelInstanceDetails>
          </bindingTemplate>

. . .

      </businessService>
    </businessServices>
    <identifierBag>
      <keyedReference keyName="D-U-N-S" keyValue="08-146-6849"
                      tModelKey="uuid:8609C81E-EE1F-4D5A-B202-3EB13AD01823"/>
    </identifierBag>
    <categoryBag>
      <keyedReference keyName="NAICS: Software Publisher" keyValue="51121"
                      tModelKey="uuid:C0B9FE13-179F-413D-8A5B-5004DB8E5BB2"/>
    </categoryBag>
  </businessEntity>
</businessDetail>
```

References

1. For a quick, very accessible introduction to UML and its graphical notation, see: Martin Fowler, *UML Distilled*, second edition, Addison-Wesley.
2. Object Management Group. *XML Metadata Interchange (XMI)*, version 1.1. See ftp://ftp.omg.org/pub/docs/ad/99-10-02.pdf
3. For current UDDI specifications, see http://www.uddi.org
4. jUDDI is an open-source Java library for UDDI, see http://www.juddi.org
5. A forthcoming book describes processes and design techniques for generating XML DTDs and Schemas from UML models. See: David Carlson, *Modeling XML Applications with UML*, Addison-Wesley, 2001 (to be published in April 2001).
6. A Web portal has been created at http://XMLModeling.com to aggregate newsfeeds and resource references related to modeling XML vocabularies, especially using UML. This site will also contain examples from the book, plus case study examples of modeling XML vocabularies.

uddi.org
Universal Description, Discovery and Integration

Exhibit C

UDDI Programmer's API 1.0
UDDI Published Specification, 28 June 2002

This version:
http://www.uddi.org/pubs/ProgrammersAPI-V1.00-Published-20020628.pdf

Latest version:
http://www.uddi.org/pubs/ProgrammersAPI-V1.00-Published-20020628.pdf

Authors (alphabetically):
Toufic Boubez, IBM
Maryann Hondo, IBM
Chris Kurt, Microsoft
Jared Rodriguez, Ariba
Daniel Rogers, Microsoft

Contents

Introduction

Document Overview

This document describes the programming interface that is exposed by all instances of the Universal Description, Discovery & Integration (UDDI) registry. The primary audience for this document is programmers that want to write software that will directly interact with a UDDI *Operator Site*[1].

What is this UDDI anyway?

UDDI is the name of a group of web-based registries that expose information about a business or other entity[2] and its technical interfaces (or API's). These registries are run by multiple *Operator Sites*, and can be used by any business that wants to make their information available, as well as anyone that wants to find that information. There is no charge for using the basic services of these operating sites.

By accessing any of the public UDDI *Operator Sites*, anyone can search for information about web services[3] that are made available by or on behalf of a business. The benefit of having access to this information is to provide a mechanism that allows others to discover what technical programming interfaces are provided for interacting with a business for such purposes as electronic commerce, etc. The benefit to the individual business is increased exposure in an electronic commerce enabled world.

The information that a business can register includes several kinds of simple data that help others determine the answers to the questions "who, what, where and how". Simple information about a business – information such as name, business identifiers (D&B D-U-N-S Number®, etc.), and contact information answers the question "*Who?*". "*What?*" involves classification information that includes industry codes and product classifications, as well as descriptive information about the services that are available for electronic interchange. Answering he question "*Where?*" involves registering information about the URL or email address (or other address) through which each type of service is accessed[4]. Finally, the question "*How?*" is answered by registering references to information about specifications that describe how a particular software package or technical interface functions. These references are called tModels in the UDDI documentation.

Compatible registries

This programmer's reference, coupled with the UDDI API schema (uddiAPI.xsd), defines a programming interface that is available for free public use. Software developers, businesses and

---

[1] *Operator Site* is a term used to describe an implementation of this specification that participates in the public cloud of UDDI sites under special contract.

[2] The term *business* is used in a general sense to refer to an operating concern or any other type of organization throughout this document. This use is not intended to preclude other organizational forms.

[3] *Web Service* is a term used to describe technical services that are exposed via some public standard. Examples include purchasing services, catalog services, search services, shipping or postal services exposed over transports like HTTP or electronic mail.

[4] The information about the service point or address at which a service is exposed is sometimes referred to using the technical term *binding information*. design specs refer to this using the term *bindingTemplate*.

others are encouraged to define products and tools that make use of this API and to build registries that are compatible with the API defined in this specification.

What are tModels?

In order for two or more pieces of software to be compatible with each other – that is, compatible enough to be able to exchange data for the purpose of achieving a desirable result – they must share some design goals and specifications in common. The registry information model that each UDDI site supports is based on this notion of shared specifications.

In the past, to build compatible software, two companies only had to agree to use the same specification, and then test their software. However, with UDDI, companies need a way to publish information about the specifications and versions of specifications that were used to design their advertised services. To accommodate the need to distinctly identify public specifications (or even private specifications shared only with select partners), information about the specifications themselves needs to be discoverable. This information about specifications – a classic metadata construct – is called a tModel within UDDI.

The tModel concept serves a useful purpose in discovering information about services that are exposed for broad use. To get a clearer understanding, let's consider an example.

An example:

Suppose your business bought a software package that let you automatically accept electronic orders via your web site. Using one of the public UDDI sites, you could advertise the availability of this electronic commerce capability.

One of the reasons you chose this particular software package was its widespread popularity. In fact the salesperson that sold you the software made a point of highlighting a feature that gives your new software its broad appeal – the use and support of a widely used set of XML business documents to accommodate automatic business data interchange.

As you installed and configured your new software, this software automatically consulted one of the public UDDI sites and identified compatible business partners. It did this by looking up each business you identified, and located those that had already advertised support for electronic commerce services that are compatible with your own.

The configuration software accomplishes this by taking advantage of the fact that a tModel has been registered for a full specification, and in the service elements for each business, the tModel keys for this specification was referenced.

In general, it's pretty safe to think of the tModel keys within a service description as a fingerprint that can be used to trace the compatibility origins of a given service. Since many services will be constructed or programmed to be compatible with a given specification, references to information about specifications (by way of tModel entries and tModel references) don't have to be repeated with each registered electronic commerce service.

For programmers that write the software that will be used by businesses, tModels provide a common point of reference that allows compatible services to be easily identified. For businesses that use this software, the benefit is greatly reduced work in determining which particular services are compatible with the software you write. Finally, for software vendors and standards organizations, the ability to register information about a specification and then find implementations of web services that are compatible with a given tModel helps customers immediately realize the benefits of a widely used design.

Classification and Identification information

One of the immediate benefits of registering business information at one of the UDDI *Operator Instances* is the ability to specify one or more classification, or category codes for your business. Many such codes exist – NAICS, UN/SPC, SIC Codes, etc. – and are widely used to classify businesses, industries, and product categories. Other (and there are many) classifications designate geographic information, or membership in a given organization.

Each UDDI site provides a way to add any number of classifications to a business registration. This information allows simple searching to be done on the information contained in the public registries. More important, registering information such as industry codes, product codes, geography codes and business identification codes (such as D&B D-U-N-S Numbers®) allow other search services to use this core classification information as a starting point to provide added-value indexing and classification while still referencing your UDDI information.

Design & Architecture

The UDDI programmer's API is designed to provide a simple request/response mechanism that allows discovery of businesses, services and technical service binding information.

Design Principles

The primary principal guiding the design of this programmers API was simplicity. Care has been taken to avoid complexity, overlap, and also to provide direct access to the appropriate levels of registered information with a minimum of programming overhead and round tripping.

Security

Accessing UDDI programmatically is accomplished via API calls defined in this programmer's reference. Two types of APIs are defined. A publishers API is provided for interactions between programs and the UDDI registry for the purpose of storing or changing data in the registry. An inquiry API is provided for programs that want to access the registry to read information from the registry.

Authenticated access is required to use the publishers API. Each *Operator Site* is responsible for selecting and implementing an authentication protocol that is compatible with the publishers API, as well as providing a new user sign-up mechanism. Before using any of the publisher API functions, the caller is responsible for signing up with one or more *Operator Sites* or compatible registries and establishing user credentials.

The Inquiry API functions are exposed as SOAP messages over HTTP protocol. No authentication is required to make use of the inquiry API functions.

Versioning

In any programmers API, as well as any message set, versioning issues arise as time passes. Changes to an API over time can result in requests being misunderstood or processed incorrectly unless one can determine whether the version of the API being provided matches the version of the API used by a requesting party.

In order to facilitate a proper and controlled version match, the entire API defined by this programmer's reference is version stamped. Since the API itself is based on XML messages transmitted in SOAP envelopes over HTTP[5], this version stamp takes the form of an XML attribute.

All of the messages defined in this API must be transmitted with an accompanying application version attribute. This attribute is named "*generic*[6]" and is present on all messages. Each time this specification is modified, an ensuing requirement is placed on all *Operator Sites* to support generic 1.0, the current generic and at least one prior generic, if any. Compatible registries are encouraged to support at a minimum this 1.0 version.

[5] HTTP is used as a general term here. HTTPS is used exclusively for all of the calls defined in the publishers API.

[6] Versioning of application behavior is accommodated via the *generic* attribute independently from the structures defined in the accompanying schema. In general, this form of versioning is preferable because it is easier to specify a new behavior against the same structures than to try and get data structure definitions to reflect business rules. Versioning the actual schema structures would present considerable technical difficulties after more than a small number of deployed applications existed.

SOAP Messaging

SOAP is a method for using Extensible Markup Language (XML) for use in message passing and remote procedure call (RPC) protocols. SOAP has been jointly defined and submitted to the World Wide Web consortium (W3C) for consideration as a standard web protocol.

SOAP is being used in conjunction with HTTP to provide a simple mechanism for passing XML messages to *Operator Sites* using a standard HTTP-POST protocol. Unless specified, all responses will be returned in the normal HTTP response document.

See the appendix on SOAP specific implementations for more information on the way that UDDI *Operator Sites* use the SOAP schema as an envelope mechanism for passing XML messages.

XML conventions

The programming interface for UDDI is based on Extensible Markup Language (XML). See the appendix (XML usage details) for more information on specific XML constructs and limitations used in the specification of the programmers interface.

Error Handling

The first line of error reporting is governed by the SOAP specification. SOAP fault reporting and fault codes will be returned for most invalid requests, or any request where the intent of the caller cannot be determined.

If any application level error occurs in processing a request message, a dispositionReport structure will be returned to the caller instead of a SOAP fault report. Disposition reports contain error information that includes descriptions and typed keys that can be used to determine the cause of the error. Refer to the appendix "Error Codes" for a general understanding of error codes. API specific interpretations of error codes are described following each API reference page.

Many of the API constructs defined in this document allow one or more of a given type of information to be passed. These API calls conceptually each represent a request on the part of the caller. The general error handling treatment is to detect errors in a request prior to processing the request. Any errors in the request detected will invalidate the entire request, and cause a dispositionReport to be generated within a SOAP Fault structure (see appendix A). In the case of an API call that involves passing multiples of a given structure, the dispositionReport will call out only the first detected error, and is not responsible for reporting multiple errors or reflecting intermediate "good" data.

White Space

Operator Sites and compatible implementations will store all data exactly as provided with one exception. Any leading or trailing white space characters will be removed from each field, element or attribute. White space characters include carriage returns, line feeds, spaces, and tabs.

XML Encoding

Despite its cross platform goals, XML still permits a broad degree of platform dependent ordering to seep into software. One of the key areas of seep has to do with the way that multiple language encoding is allowable in the XML specification. For the purpose of this specification and all UDDI *Operator Sites* consistency in handling of data is essential. For this reason, the default collation order for data registered within an *Operator Site* is binary. See appendix B for more information related to the use of byte order marks and UTF-8 and the way the SOAP listeners convert all requests to Unicode prior to processing.

API Reference

This API reference is divided into n logical sections. Each section addresses a particular programming focus. The sections are arranged in order according to the most common uses, and within each section alphabetically.

Special values within API syntax examples are shown in italics. In most cases, the following reference applies to these values:

- *uuid_key*: Access keys within all UDDI defined data elements are represented as universal unique identifiers (these are sometimes called a GUID). The name of the element or attribute designates the particular key type that is required. These keys are always formatted according to DCE UUID conventions with the one exception being tModelKey values, which are prefixed with a URN qualifier in the format "uuid:" followed by the UUID value.

- *generic*: This special attribute is a required metadata element for all messages. It is used to designate the specification version used to format the SOAP message. In the 1.0 version of the specification, this value is required to be "1.0". Any other value passed can result in an E_unsupported error.

- *xmlns*: This special attribute is a required metadata element for all messages. Technically, it isn't an attribute, but is formally called a namespace qualifier. It is used to designate is a universal resource name (URN) value that is reserved for all references to the UDDI schema. In the 1.0 version of the specification, this value is required to be "urn:uddi-org:api".

- *findQualifiers*: This special element is found in the inquiry API functions that are used to search (e.g. find_binding, find_business, find_service, find_tModel). This argument is used to signal special behaviors to be used with searching. See the Search Qualifiers appendix for more information.

- *maxRows*: This special qualifier is found in the inquiry API functions that are used to search (e.g. find_binding, find_business, find_service, find_tModel). This argument is used to limit the number of results returned from a request. When an *Operator Site* or compatible instance returns data in response to a request that contains this limiting argument, the number of results will not exceed the integer value passed. If a result set is truncated as a result of applying this limit, the result will include the *truncated* attribute with a value of *true*.

- *truncated*: The truncated attribute indicates that a maximum number of possible results has been returned. The actual limit set for applying this treatment is *Operator Site* policy specific, but in general should be a sufficiently large number so as to not normally be an issue. No behaviors such as paging mechanisms are defined for retrieving more data after a truncated limit. The intent is to support the average query, but to allow *Operator Sites* the leeway required to be able to manage adequate performance.

- *categoryBag*: Searches can be performed based on a cross section of categories. Several categories are broadly supported by all *Operator Sites* and provide three categorization dimensions. These are industry type, product and service type, and geography. Searches involving category information can be combined to cross multiple dimensions. For this reason, these searches are performed matching on ALL of the categories supplied. The net effect in generic 1 is the ability to use embedded category information as hints about how the registering party has categorized themselves, but not to provide a full third party categorization facility. This is the realm of portals and marketplaces, and may be enhanced in future generics.

- *identifierBag:* Searches involving identifiers are performed matching on any supplied identifier (e.g. D&B D-U-N-S Number®, etc) for any of the primary elements that have identifierBag elements. These searches allow broad identity matching by returning a match when any keyedReference set used to search identifiers matches a registered identifier.

- *tModelBag:* Searches that match a particular technical fingerprint use UUID values to search for bindingTemplates with matching tModelKey value sets. When used to search for web services (e.g. the data described by a bindingTemplate structure), the concept of tModel signatures allows for highly selective searches for specific combinations of keys. For instance, the existence of a web service that implements all of the parts of the UDDI specifications can be accomplished by searching for a combination of tModel key values that correspond to the full set of specifications (the UDDI specification, for instance, is divided into at least 5 different, separately deployable tModels). At the same time, limiting the number of tModelKey values passed in a search can perform broader searches that look for any web service that implements a specific sub-part of the full specification. All tModelKey values are always expressed using a Universal Resource Name (URN) format that starts with the characters "uuid:" followed by a formatted Universally Unique Identifier (UUID) consisting of an octet of Hexidecimal digits arranged in the common 12-4-4-8 format pattern.

In all cases, the XML structures, attributes and element names shown in the API examples are derived from the Message API schema. For a full understanding of structure contents, refer to this schema. It is suggested that tools that understand schemas be used to generate logic that populates the structures used to make the API calls against an *Operator Site*.

Three query patterns

The Inquiry API provides three forms of query that follow broadly used conventions. These two forms match the needs of two types of software that are traditionally used with registries.

The browse pattern

Software that allows people to explore and examine data – especially hierarchical data – requires browse capabilities. The browse pattern characteristically involves starting with some broad information, performing a search, finding general result sets and then selecting more specific information for drill-down.

The UDDI API specifications accommodate the browse pattern by way of the *find_xx* API calls. These calls form the search capabilities provided by the API and are matched with summary return messages that return overview information about the registered information that match the supplied search criteria.

A typical browse sequence might involve finding whether a particular business you know about has any information registered. This sequence would start with a call to *find_business*, perhaps passing the first few characters of the businesses name that you already know. This returns a businessList result. This result is overview information (keys, names and descriptions) of the businessEntity information that matched the search results returned by find_business.

If you spot the business you are looking for, you can drill into their businessService information, looking for particular service types (e.g. purchasing, shipping, etc) using the find_service API call. Similarly, if you know the technical fingerprint (tModel signature) of a particular product and want to see if the business you've chosen supports a compatible service interface, you can use find_binding.

The drill-down pattern

Once you have a key for one of the four main data types managed by a UDDI or compatible registry[7], you can use that key to access the full registered details for a specific data instance. The current UDDI data types are businessEntity, businessService, bindingTemplate and tModel. You can access the full registered information for any of these structures by passing a relevant key type to one of the *get_xx* API calls.

Continuing the example from the previous section on browsing, one of the data items returned by all of the *find_xx* return sets is key information. In the case of the business we were interested in, the businessKey value returned within the contents of a businessList structure can be passed as an argument to *get_businessDetail*. The successful return to this message is a *businessDetail* message containing the full registered information for the entity whose key value was passed. This will be a full *businessEntity* structure.

The invocation pattern

In order to prepare an application to take advantage of a remote web service that is registered within the UDDI registry by other businesses or entities, you need to prepare that application to use the information found in the registry for the specific service being invoked. This type of cross business service call has traditionally been a task that is undertaken at development time. This will not necessarily change completely as a result of UDDI registry entries, but one significant problem can be managed if a particular invocation pattern is employed.

Data obtained from the UDDI registry about a *bindingTemplate* information set represents the instance specifics of a given remote service. The program should cache this information and use it to contact the service at the registered address. Tools have automated the tasks associated with caching (or hard coding) location information in previously popular remote procedure technologies. Problems arise however when a remote service is moved without any knowledge on the part of the callers. Moves occur for a variety of reasons, including server upgrades, disaster recovery, and service acquisition and business name changes.

When a call fails using cached information obtained from a UDDI registry, the proper behavior is to query the UDDI registry where the data was obtained for fresh bindingTemplate information. The proper call is *get_bindingDetails* passing the original bindingKey value. If the data returned is different from the cached information, the service invocation should automatically retry. If the result of this retry is successful, the new information should replace the cached information.

By using this pattern with web services, a business using a UDDI *Operator Site* can automate the recovery of a large number of partners without undue communication and coordination costs. For example, if a business has activated a disaster recovery site, most of the calls from partners will fail when they try to invoke services at the failed site. By updating the UDDI information with the new address for the service, partners who use the invocation pattern will automatically locate the new service information and recover without further administrative action.

[7] Keys within UDDI compatible registries that are not *Operator Sites* are not synchronized with keys generated by *Operator Sites*. There is no key portability mechanism defined for crossing from a replicated operator site to a compatible registry that is not part of the replicated *Operator Cloud*.

Inquiry API functions

The messages in this section represent inquiries that anyone can make of any UDDI *Operator Site* at any time. These messages all behave synchronously and are required to be exposed via HTTP-POST only. Other synchronous or asynchronous mechanisms may be provided at the discretion of the individual UDDI *Operator Site* or UDDI compatible registry.

The publicly accessible queries are:

- **find_binding**: Used to locate specific bindings within a registered businessService. Returns a bindingDetail message.
- **find_business**: Used to locate information about one or more businesses. Returns a businessList message.
- **find_service**: Used to locate specific services within a registered businessEntity. Returns a serviceList message.
- **find_tModel**: Used to locate one or more tModel information structures. Returns a tModelList structure.
- **get_bindingDetail**: Used to get full bindingTemplate information suitable for making one or more service requests. Returns a bindingDetail message.
- **get_businessDetail**: Used to get the full businessEntity information for a one or more businesses. Returns a businessDetail message.
- **get_businessDetailExt**: Used to get extended businessEntity information. Returns a businessDetailExt message.
- **get_serviceDetail**: Used to get full details for a given set of registered businessService date. Returns a serviceDetail message.
- **get_tModelDetail**: Used to get full details for a given set of registered tModel data. Returns a tModelDetail message.

find_binding

The find_binding message returns a bindingDetail message that contains a *bindingTemplates* structure with zero or more bindingTemplate structures matching the criteria specified in the argument list.

Syntax:

```
<find_binding serviceKey="uuid_key" generic="1.0" [ maxRows="nn" ]
      xmlns="urn:uddi-org:api" >
   [<findQualifiers/>]
   <tModelBag/>
</find_binding>
```

Arguments:

- ***maxRows***: This optional integer value allows the requesting program to limit the number of results returned.
- ***serviceKey***: This *uuid_key* is used to specify a particular instance of a businessService element in the registered data. Only bindings in the specific businessService data identified by the serviceKey passed will be searched.
- ***findQualifiers***: This collection of findQualifier elements can be used to alter the default behavior of search functionality. See the Search Qualifiers appendix for more information.
- ***tModelBag***: This is a list of tModel *uuid_key* values that represent the technical fingerprint to locate in a bindingTemplate structure contained within the businessService instance specified by the serviceKey value. If more than one tModel key is specified in this structure, only bindingTemplate information that exactly matches all of the tModel keys specified will be returned (logical AND). The order of the keys in the tModel bag is not relevant. All tModelKey values begin with a uuid URN qualifier (e.g. "uuid:" followed by a known tModel UUID value.

Returns:

This function returns a bindingDetail message on success. In the event that no matches were located for the specified criteria, the bindingDetail structure returned in the response the will be empty (e.g. contain no bindingTemplate data.)

In the even of a large number of matches, an *Operator Site* may truncate the result set. If this occurs, the response message will contain the *truncated* attribute with the value of this attribute set to *true*.

Searching using tModelBag will also return any bindingTemplate information that matches due to hostingRedirector references. The resolved bindingTemplate structure will be returned, even if that bindingTemplate is owned by a different businessService structure.

Caveats:

If any error occurs in processing this message, a dispositionReport structure will be returned to the caller in a SOAP Fault. The following error number information will be relevant:

- **E_invalidKeyPassed**: signifies that the *uuid_key* value passed did not match with any known serviceKey key or tModel key values. The error structure will signify which condition occurred first.
- **E_tooManyOptions**: signifies that more than one mutually exclusive argument was passed.
- **E_unsupported**: signifies that one of the findQualifier values passed was invalid.

find_business

The find_business message returns a businessList message that matches the conditions specified in the arguments.

Syntax:

```
<find_business  generic="1.0" [ maxRows="nn" ] xmlns="urn:uddi-org:api" >
    [<findQualifiers/>]
    <name/> | <identifierBag/> | <categoryBag/> | <tModelBag/> | <discoveryURLs>
</find_business>
```

Arguments: All arguments to this call listed are mutually exclusive except findQualifiers

- ***maxRows***: This optional integer value allows the requesting program to limit the number of results returned.
- ***findQualifiers***: This collection of findQualifier elements can be used to alter the default behavior of search functionality. See the Search Qualifiers appendix for more information.
- ***name***: This string value is a partial name. The businessList return contains businessInfo structures for businesses whose name matches the value passed (leftmost match).
- ***identifierBag***: This is a list of business identifier references. The returned businessList contains businessInfo structures matching any of the identifiers passed (logical OR).
- ***categoryBag***: This is a list of category references. The returned businessList contains businessInfo structures matching all of the categories passed (logical AND).
- ***tModelBag***: The registered businessEntity data contains bindingTemplates that in turn contain specific tModel references. The tModelBag argument lets you search for businesses that have bindings that are compatible with a specific tModel pattern. The returned businessList contains businessInfo structures that match all of the tModel keys passed (logical AND). tModelKey values must be formatted as URN qualified UUID values (e.g. prefixed with "uuid:")
- ***discoveryURLs***: This is a list of URL's to be matched against the data associated with the discoveryURL's contents of registered businessEntity information. To search for URL without regard to useType attribute values, pass the useType component of the discoveryURL elements as empty attributes. If useType values are included, then the match will be made only on registered information that match both the useType and URL value. The returned businessList contains businessInfo structures matching any of the URL's passed (logical OR).

Returns:

This function returns a businessList on success. In the event that no matches were located for the specified criteria, a businessList structure with zero businessInfo structures is returned.

In the event of a large number of matches, an *Operator Site* may truncate the result set. If this occurs, the businessList will contain the *truncated* attribute with the value set to *true*.

Searching using tModelBag will also return any businessEntity that contains bindingTemplate information that matches due to hostingRedirector references. In other words, the businessEntity that contains a bindingTemplate with a hostingRedirector value referencing a bindingTemplate that matches the tModel search requirements will be returned.

Caveats:

If any error occurs in processing this message, a dispositionReport structure will be returned to the caller in a SOAP Fault. The following error number information will be relevant:

- **E_nameTooLong**: signifies that the partial name value passed exceeds the maximum name length designated by the *Operator Site*.
- **E_tooManyOptions**: signifies that more than one search argument was passed.
- **E_unsupported**: signifies that one of the findQualifier values passed was invalid.

find_service

The find_service message returns a serviceList message that matches the conditions specified in the arguments.

Syntax:

```
<find_service businessKey="uuid_key" generic="1.0"  [ maxRows="nn" ]
        xmlns="urn:uddi-org:api" >
     [<findQualifiers/>]
     <name/> | <categoryBag/> | <tModelBag/>
</find_service>
```

Arguments: The *name*, categoryBag and *tModelBag* arguments are mutually exclusive

- ***maxRows***: This optional integer value allows the requesting program to limit the number of results returned:
- ***businessKey***: This *uuid_key* is used to specify a particular BusinessEntity instance.
- ***findQualifiers***: This collection of findQualifier elements can be used to alter the default behavior of search functionality. See the Search Qualifiers appendix for more information.
- ***name***: This string value represents a partial name. Any businessService data contained in the specified businessEntity with a matching partial name value gets returned.
- ***categoryBag***: This is a list of category references. The returned serviceList contains businessInfo structures matching all of the categories passed (logical AND).
- ***tModelBag***: This is a list of tModel *uuid_key* values that represent the technical fingerprint to locate within a bindingTemplate structure contained within any businessService contained by the businessEntity specified. If more than one tModel key is specified in this structure, only businessServices that contain bindingTemplate information that matches all of the tModel keys specified will be returned (logical AND)

Returns:

This function returns a serviceList on success. In the event that no matches were located for the specified criteria, the serviceList structure returned will contain an empty businessServices structure. This signifies zero matches.

In the even of a large number of matches, an *Operator Site* may truncate the result set. If this occurs, the serviceList will contain the *truncated* attribute with the value of this attribute set to *true*.

Searching using tModelBag will return serviceInfo structure for all qualifying businesService data, including matches due to hostingRedirector references. In other words, if the businessEntity whose businessKey is passed as an argument contains a bindingTemplate with a hostingRedirector value, and that value references a bindingTemplate that matches the tModel search requirements, then the serviceInfo for the businessService containing the hostingRedirector will be returned.

Caveats:

If any error occurs in processing this message, a dispositionReport structure will be returned to the caller in a SOAP Fault. The following error number infomation will be relevant:

- **E_invalidKeyPassed**: signifies that the *uuid_key* value passed did not match with any known businessKey key or tModel key values. The error structure will signify which condition occurred first.
- **E_nameTooLong**: signifies that the partial name value passed exceeds the maximum name length designated by the *Operator Site*.
- **E_tooManyOptions**: signifies that more than one mutually exclusive argument was passed.
- **E_unsupported**: signifies that one of the findQualifier values passed was invalid.

find_tModel

This find_tModel message is for locating a list of tModel entries that match a set of specific criteria. The response will be a list of abbreviated information about tModels that match the criteria (tModelList).

Syntax:

```
<find_tModel generic="1.0" [ maxRows="nn" ]   xmlns="urn:uddi-org:api" >
    [<findQualifiers/>]
    <name/> | <identifierBag/> | <categoryBag/>
</find_tModel>
```

Arguments: The arguments to this call are mutually exclusive except findQualifiers

- ***maxRows***: This optional integer value allows the requesting program to limit the number of results returned.
- ***findQualifiers***: This collection of findQualifier elements can be used to alter the default behavior of search functionality. See the Search Qualifiers appendix for more information.
- ***name:*** This string value represents a partial name. The returned tModelList contains tModelInfo structures for businesses whose name matches the value passed (leftmost match).
- ***IdentifierBag***: This is a list of business identifier references. The returned tModelList contains tModelInfo structures matching any of the identifiers passed (logical OR).
- ***categoryBag***: This is a list of category references. The returned tModelList contains tModelInfo structures matching all of the categories passed (logical AND).

Returns:

This function returns a tModelList on success. In the event that no matches were located for the specified criteria, an empty tModelList structure will be returned (e.g. will contain zero tModelInfo structures). This signifies zero matches.

In the even of a large number of matches, an *Operator Site* may truncate the result set. If this occurs, the tModelList will contain the *truncated* attribute with the value of this attribute set to *true*.

Caveats:

If any error occurs in processing this message, a dispositionReport structure will be returned to the caller in a SOAP Fault. The following error number information will be relevant:

- **E_nameTooLong**: signifies that the partial name value passed exceeds the maximum name length designated by the *Operator Site*.
- **E_tooManyOptions**: signifies that more than one mutually exclusive argument was passed.
- **E_unsupported**: signifies that one of the findQualifier values passed was invalid.

get_bindingDetail

The get_bindingDetail message is for requesting the run-time bindingTemplate information location information for the purpose of invoking a registered business API.

Syntax:

```
<get_bindingDetail  generic="1.0" xmlns="urn:uddi-org:api" >
    <bindingKey/>
    [ <bindingKey/> ...]
</get_bindingDetail>
```

Arguments:

- ***bindingKey*** : one or more *uuid_key* values that represent specific instances of known bindingTemplate data.

Behavior:

In general, it is recommended that bindingTemplate information be cached locally by applications so that repeated calls to a service described by a bindingTemplate can be made without having to make repeated round trips to an UDDI registry. In the event that a call made with cached data fails, the get_bindingDetail message can be used to get fresh bindingTemplate data. This is useful in cases such as a service you are using relocating to another server or being restored in a disaster recovery site.

Returns:

This function returns a bindingDetail message on successful match of one or more bindingKey values. If multiple bindingKey values were passed, the results will be returned in the same order as the keys passed.

In the even of a large number of matches, an *Operator Site* may truncate the result set. If this occurs, the bindingDetail result will contain the *truncated* attribute with the value of this attribute set to *true*.

Caveats:

If any error occurs in processing this message, a dispositionReport structure will be returned to the caller in a SOAP Fault. The following error number information will be relevant:

- **E_invalidKeyPassed**: signifies that one of the *uuid_key* values passed did not match with any known bindingKey key values. No partial results will be returned – if any bindingKey values passed are not valid bindingKey values, this error will be returned.

get_businessDetail

The get_businessDetail message returns complete businessEntity information for one or more specified businessEntitys.

Syntax:

```
<get_businessDetail generic="1.0" xmlns="urn:uddi-org:api" >
    <businessKey/>
    [ <businessKey/> ...]
</get_businessDetail>
```

Arguments:

- ***businessKey*** : one or more *uuid_key* values that represent specific instances of known businessEntity data.

Returns:

This function returns a businessDetail message on successful match of one or more businessKey values. If multiple businessKey values were passed, the results will be returned in the same order as the keys passed.

In the even of a large number of matches, an *Operator Site* may truncate the result set. If this occurs, the businessDetail response message will contain the *truncated* attribute with the value of this attribute set to *true*.

Caveats:

If any error occurs in processing this message, a dispositionReport structure will be returned to the caller in a SOAP Fault. The following error number information will be relevant:

- **E_invalidKeyPassed:** signifies that one of the *uuid_key* values passed did not match with any known businessKey values. No partial results will be returned – if any businessKey values passed are not valid, this error will be returned.

get_businessDetailExt

The get_businessDetailExt message returns extended businessEntity information for one or more specified businessEntitys. This message returns exactly the same information as the get_businessDetail message, but may contain additional attributes if the source is an external registry (not an *Operator Site*) that is compatible with this API specification.

Syntax:

```
<get_businessDetailExt generic="1.0" xmlns="urn:uddi-org:api" >
   <businessKey/>
   [ <businessKey/> …]
</get_businessDetailExt>
```

Arguments:

- ***businessKey*** : one or more *uuid_key* values that represent specific instances of known businessEntity data.

Returns:

This function returns a businessDetailExt message on successful match of one or more businessKey values. If multiple businessKey values were passed, the results will be returned in the same order as the keys passed.

In the even of a large number of matches, an *Operator Site* may truncate the result set. If this occurs, the businessDetailExt response message will contain the *truncated* attribute with the value of this attribute set to *true*.

Caveats:

If any error occurs in processing this message, a dispositionReport structure will be returned to the caller in a SOAP Fault. The following error number information will be relevant:

- **E_invalidKeyPassed**: signifies that one of the *uuid_key* values passed did not match with any known businessKey values. No partial results will be returned – if any businessKey values passed are not valid, this error will be returned.

- **E_unsupported**: signifies that the implementation queried does not support the extended detail function. If this occurs, businessDetail information should be queried via the get_businessDetail API. *Operator Sites* will not return this code, but will instead return a businessDetailExt result with full businessDetail information embedded.

get_serviceDetail

The get_serviceDetail message is used to request full information about a known businessService structure.

Syntax:

```
<get_serviceDetail generic="1.0" xmlns="urn:uddi-org:api" >
    <serviceKey/>
    [ <serviceKey/> ...]
</get_serviceDetail>
```

Arguments:

- ***serviceKey*** : one or more *uuid_key* values that represent specific instances of known businessService data.

Returns:

This function returns a serviceDetail message on successful match of one or more serviceKey values. If multiple serviceKey values were passed, the results will be returned in the same order as the keys passed.

In the even of a large number of matches, an *Operator Site* may truncate the result set. If this occurs, the response will contain a *truncated* attribute with the value of this attribute set to *true*.

Caveats:

If any error occurs in processing this message, a dispositionReport structure will be returned to the caller in a SOAP Fault. The following error number information will be relevant:

- **E_invalidKeyPassed**: signifies that one of the *uuid_key* values passed did not match with any known serviceKey values. No partial results will be returned – if any serviceKey values passed are not valid, this error will be returned.

get_tModelDetail

The get_tModelDetail message is used to request full information about a known tModel structure.

Syntax:

```
<get_tModelDetail  generic="1.0" xmlns="urn:uddi-org:api" >
    <tModelKey/>
    [ <tModelKey/> ...]
</get_tModelDetail>
```

Arguments:

- ***tModelKey*** : one or more URN qualified *uuid_key* values that represent specific instances of known tModel data. All tModelKey values begin with a uuid URN qualifier (e.g. "uuid:" followed by a known tModel UUID value.)

Returns:

This function returns a tModelDetail message on successful match of one or more tModelKey values. If multiple tModelKey values were passed, the results will be returned in the same order as the keys passed.

In the even of a large number of matches, an *Operator Site* may truncate the result set. If this occurs, the response will contain a *truncated* attribute with the value of this attribute set to *true*.

Caveats:

If any error occurs in processing this message, a dispositionReport structure will be returned to the caller in a SOAP Fault. The following error number information will be relevant:

- **E_invalidKeyPassed**: signifies that one of the URN qualified *uuid_key* values passed did not match with any known tModelKey values. No partial results will be returned – if any tModelKey values passed are not valid, this error will be returned. Any tModelKey values passed without a uuid URN qualifier will be considered invalid.

- **E_keyRetired**: signifies that the request cannot be satisfied because the owner has retired the tModel information. The tModel reference may still be valid and used as intended, but the information defining the tModel behind the key is unavailable.

Publishing API functions

The messages in this section represent inquiries that require authenticated[8] access to an UDDI *Operator Site*. Each business should initially select one *Operator Site* to host their information. Once chosen, information can only be updated at the site originally selected.

The messages defined in this section all behave synchronously and are callable via HTTP-POST only. HTTPS is used exclusively for all of the calls defined in this publishers API.

The publishing API calls are:

- **delete_binding**: Used to remove an existing bindingTemplate from the bindingTemplates collection that is part of a specified businessService structure.
- **delete_business**: Used to delete registered businessEntity information from the registry.
- **delete_service**: Used to delete an existing businessService from the businessServices collection that is part of a specified businessEntity.
- **delete_tModel**: Used to delete registered information about a tModel. If there are any references to a tModel when this call is made, the tModel will be marked deleted instead of being physically removed.
- **discard_authToken**: Used to inform an *Operator Site* that a previously provided authentication token is no longer valid. See get_authToken.
- **get_authToken**: Used to request an authentication token from an *Operator Site*. Authentication tokens are required to use all other API's defined in the publishers API. This function serves as the programs equivalent of a login request.
- **get_registeredInfo**: Used to request an abbreviated synopsis of all information currently managed by a given individual.
- **save_binding**: Used to register new bindingTemplate information or update existing bindingTemplate information. Use this to control information about technical capabilities exposed by a registered business.
- **save_business**:. Used to register new businessEntity information or update existing businessEntity information. Use this to control the overall information about the entire business. Of the save_x API's this one has the broadest effect.
- **save_service**: Used to register or update complete information about a businessService exposed by a specified businessEntity.
- **save_tModel**: Used to register or update complete information about a tModel.

[8] Authentication is not regulated by this API specification. Individual *Operator Sites* will designate their own procedures for getting a userID and password.

Special considerations around categorization

Several of the API's defined in this section allow you to save categorization information that is used to support searches that use taxonomy references. These are currently the save_business, save_service and save_tModel APIs. Categorization is specified using an optional element named categoryBag, which contains namespace-qualified references to taxonomy keys and descriptions in keyedReference structures.

Data contained in the keyValue attribute of each keyedReference is validated against the taxonomy referenced by the associated tModelKey. Only valid keyValues will be stored as entered. *Operator Sites* may handle invalid keyValues by either fail the request or changing the tModelKey value to reference an non-validated "etc." taxonomy which accepts all keyValues. See Appendix H for details on the validation of taxonomic information. If the *Operator Site* chooses to fail categorization mechanisms, the error codes defined in appendix H will be passed back as the error code on the relevant API calls.

delete_binding

The delete_binding message causes one or more bindingTemplate to be deleted.

Syntax:

```
<delete_binding generic="1.0" xmlns="urn:uddi-org:api" >
    <authInfo/>
    <bindingKey/>
    [ <bindingKey/> ...]
</delete_binding>
```

Arguments:

- ***authInfo***: this required argument is an element that contains an authentication token. Authentication tokens are obtained using the get_authToken API call.
- ***bindingKey*** : one or more *uuid_key* values that represent specific instances of known bindingTemplate data.

Returns:

Upon successful completion, a dispositionReport is returned with a single success indicator.

Caveats:

If any error occurs in processing this message, a dispositionReport structure will be returned to the caller in a SOAP Fault. The following error number information will be relevant:

- **E_invalidKeyPassed**: signifies that one of the *uuid_key* values passed did not match with any known bindingKey values. No partial results will be returned – if any bindingKey values passed are not valid, this error will be returned.
- **E_authTokenExpired**: signifies that the authentication token value passed in the authInfo argument is no longer valid because the token has expired.

- **E_authTokenRequired**: signifies that the authentication token value passed in the authInfo argument is either missing or is not valid.
- **E_userMismatch**: signifies that one or more of the bindingKey values passed refers to data that is not controlled by the individual who is represented by the authentication token.
- **E_operatorMismatch**: signifies that one or more of the bindingKey values passed refers to data that is not controlled by the *Operator Site* that received the request for processing.

delete_business

The delete_business message is used to remove one or more businessEntity structures.

Syntax:

```
<delete_business  generic="1.0" xmlns="urn:uddi-org:api" >
    <authInfo/>
    <businessKey/>
    [ <businessKey/> ...]
</delete_business>
```

Arguments:

- ***authInfo***: this required argument is an element that contains an authentication token. Authentication tokens are obtained using the get_authToken API call.
- ***businessKey*** : one or more *uuid_key* values that represent specific instances of known businessEntity data.

Returns:

Upon successful completion, a dispositionReport is returned with a single success indicator.

Caveats:

If any error occurs in processing this message, a dispositionReport structure will be returned to the caller in a SOAP Fault. The following error number information will be relevant:

- **E_invalidKeyPassed**: signifies that one of the *uuid_key* values passed did not match with any known businessKey values. No partial results will be returned – if any businessKey values passed are not valid, this error will be returned.
- **E_authTokenExpired**: signifies that the authentication token value passed in the authInfo argument is no longer valid because the token has expired.
- **E_authTokenRequired**: signifies that the authentication token value passed in the authInfo argument is either missing or is not valid.
- **E_userMismatch**: signifies that one or more of the businessKey values passed refers to data that is not controlled by the individual who is represented by the authentication token.
- **E_operatorMismatch**: signifies that one or more of the businessKey values passed refers to data that is not controlled by the *Operator Site* that received the request for processing.

delete_service

The delete_service message is used to remove one or more businessService structures.

Syntax:

```
<delete_service  generic="1.0" xmlns="urn:uddi-org:api" >
    <authInfo/>
    <serviceKey/>
    [ <serviceKey/> …]
</delete_service>
```

Arguments:

- ***authInfo***: this required argument is an element that contains an authentication token. Authentication tokens are obtained using the get_authToken API call.
- ***serviceKey*** : one or more *uuid_key* values that represent specific instances of known businessService data.

Returns:

Upon successful completion, a dispositionReport is returned with a single success indicator.

Caveats:

If any error occurs in processing this message, a dispositionReport structure will be returned to the caller in a SOAP Fault. The following error number information will be relevant:

- **E_invalidKeyPassed**: signifies that one of the *uuid_key* values passed did not match with any known serviceKey values. No partial results will be returned – if any serviceKey values passed are not valid, this error will be returned.
- **E_authTokenExpired**: signifies that the authentication token value passed in the authInfo argument is no longer valid because the token has expired.
- **E_authTokenRequired**: signifies that the authentication token value passed in the authInfo argument is either missing or is not valid.
- **E_userMismatch**: signifies that one or more of the serviceKey values passed refers to data that is not controlled by the individual who is represented by the authentication token.
- **E_operatorMismatch**: signifies that one or more of the serviceKey values passed refers to data that is not controlled by the *Operator Site* that received the request for processing.

delete_tModel

The delete_tModel message is used to remove or retire one or more tModel structures.

Syntax:

```
<delete_tModel generic="1.0" xmlns="urn:uddi-org:api" >
    <authInfo/>
    <tModelKey/> [ <tModelKey/> ...]
</delete_tModel>
```

Arguments:

- ***authInfo***: this required argument is an element that contains an authentication token. Authentication tokens are obtained using the get_authToken API call.

- ***tModelKey*** : one or more URN qualified *uuid_key* values that represent specific instances of known tModel data. All tModelKey values begin with a uuid URN qualifier (e.g. "uuid:" followed by a known tModel UUID value.)

Returns:

Upon successful completion, a dispositionReport is returned with a single success indicator.

Behavior:

If a tModel is deleted and any other managed data references to that tModel by *uuid_key* (e.g. within a categoryBag, identifierBag or within a tModelInstanceInfo structure) it will not be physically deleted as a result of this call. Instead it will be marked as hidden. Any tModels hidden in this way are still accessible to their owner, via the get_registeredInfo, but will be omitted from any results returned by calls to find_tModel. The details associated with a hidden tModel are still available to anyone that uses the get_tModelDetail message. Publishing parties that want to remove all details about a tModel from the system should call save_tModel, passing empty values in the data fields, before calling this function. A hidden tModel can be restored and made universally visible by invoking the save_tModel API at a later time, passing the key of the hidden tModel.

Caveats:

If any error occurs in processing this message, a dispositionReport structure will be returned to the caller in a SOAP Fault. The following error number information will be relevant:

- **E_invalidKeyPassed**: signifies that one of the URN qualified *uuid_key* values passed did not match with any known tModelKey values. No partial results will be returned – if any tModelKey values passed are not valid, this error will be returned. Any tModelKey values passed without a uuid URN qualifier will be considered invalid.

- **E_authTokenExpired**: signifies that the authentication token value passed in the authInfo argument is no longer valid because the token has expired.

- **E_authTokenRequired**: signifies that the authentication token value passed in the authInfo argument is either missing or is not valid.

- **E_userMismatch**: signifies that one or more of the tModelKey values passed refers to data that is not controlled by the individual who is represented by the authentication token.

- **E_operatorMismatch**: signifies that one or more of the tModelKey values passed refers to data that is not controlled by the *Operator Site* that received the request for processing.

discard_authToken

The discard_authToken message is used to inform an *Operator Site* that the authentication token can be discarded. Subsequent calls that use the same authToken may be rejected. This message is optional for *Operator Sites* that do not manage session state or that do not support the get_authToken message.

Syntax:

```
<discard_authToken generic="1.0" xmlns="urn:uddi-org:api" >
    <authInfo/>
</discard_authToken>
```

Arguments:

- ***authInfo***: this required argument is an element that contains an authentication token. Authentication tokens are obtained using the get_authToken API call.

Returns:

Upon successful completion, a dispositionReport is returned with a single success indicator. Discarding an expired authToken will be processed and reported as a success condition.

Caveats:

If any error occurs in processing this message, a dispositionReport structure will be returned to the caller in a SOAP Fault. The following error number information will be relevant:

- **E_authTokenRequired**: signifies that the authentication token value passed in the authInfo argument is either missing or is not valid.

get_authToken

The get_authToken message is used to obtain an authentication token. Authentication tokens are opaque values that are required for all other publisher API calls. This message is not required for *Operator Sites* that have an external mechanism defined for users to get an authentication token. This API is provided for implementations that do not have some other method of obtaining an authentication token or certificate, or that choose to use userID and Password based authentication.

Syntax:

```
<get_authToken  generic="1.0" xmlns="urn:uddi-org:api"
    userID="someLoginName"
    cred="someCredential"
</get_authToken>
```

Arguments:

- *userID*: this required attribute argument is the user that an individual authorized user was assigned by an *Operator Site*. *Operator Sites* will each provide a way for individuals to obtain a UserID and password that will be valid only at the given *Operator Site*.
- *cred*: this required attribute argument is the password or credential that is associated with the user.

Returns:

This function returns an authToken message that contains a valid authInfo element that can be used in subsequent calls to publisher API calls that require an authInfo value.

Caveats:

If any error occurs in processing this message, a dispositionReport structure will be returned to the caller in a SOAP Fault. The following error number information will be relevant:

- **E_unknownUser**: signifies that the Operator Site that received the request does not recognize the userID and/or pwd argument values passed as valid credentials.

get_registeredInfo

The get_registeredInfo message is used to get an abbreviated list of all businessEntity keys and tModel keys that are controlled by the individual associated the credentials passed.

Syntax:

```
<get_registeredInfo  generic="1.0" xmlns="urn:uddi-org:api" >
    <authInfo/>
</get_registeredInfo>
```

Arguments:

- ***authInfo***: this required argument is an element that contains an authentication token. Authentication tokens are obtained using the get_authToken API call.

Returns:

Upon successful completion, a registeredInfo structure will be returned, listing abbreviated business information in one or more businessInfo structures, and tModel information in one or more tModelInfo structures. This API is useful for determining the full extent of registered information controlled by a single user in a single call.

Caveats:

If any error occurs in processing this message, a dispositionReport structure will be returned to the caller in a SOAP Fault. The following error number information will be relevant:

- **E_authTokenExpired**: signifies that the authentication token value passed in the authInfo argument is no longer valid because the token has expired.
- **E_authTokenRequired**: signifies that the authentication token value passed in the authInfo argument is either missing or is not valid.

save_binding

The save_binding message is used to save or update a complete bindingTemplate structure. This message can be used to add or update one or more bindingTemplate structures to one or more existing businessService structures.

Syntax:

```
<save_binding generic="1.0" xmlns="urn:uddi-org:api" >
    <authInfo/>
    <bindingTemplate/> [<bindingTemplate/>...]
</save_binding>
```

Arguments:

- ***authInfo***: this required argument is an element that contains an authentication token. Authentication tokens are obtained using the get_authToken API call.

- ***bindingTemplate***: one or more complete bindingTemplate structures. The order in which these are processed is not defined. To save a new bindingTemplate, pass a bindingTemplate structure with an empty bindingKey attribute value.

Behavior:

Each bindingTemplate structure passed must contain a serviceKey value that corresponds to a registered businessService controlled by the same person saving the bindingTemplate data. The net effect of this call is to establish the parent businessService relationship for each bindingTemplate affected by this call. If the same bindingTemplate (determined by matching bindingKey value) is listed more than once, any relationship to the containing businessService will be determined by processing order, which is determined by the position of the bindingTemplate data in first to last order.

Using this message it is possible to move an existing bindingTemplate structure from one businessService structure to another by simply specifying a different parent businessService relationship. Changing a parent relationship in this way will cause two businessService structures to be affected.

If a bindingTemplate being saved contains a hostingRedirector element, and that element references a bindingTemplate that itself contains a hostingRedirector element, an error condition (E_invalidKeyPassed) will be generated.

Returns:

This API returns a bindingDetail message containing the final results of the call that reflects the newly registered information for the effected bindingTemplate structures.

Caveats:

If any error occurs in processing this message, a dispositionReport structure will be returned to the caller in a SOAP Fault. The following error number information will be relevant:

- **E_authTokenExpired**: signifies that the authentication token value passed in the authInfo argument is no longer valid because the token has expired.

- **E_authTokenRequired**: signifies that the authentication token value passed in the authInfo argument is either missing or is not valid.
- **E_keyRetired**: signifies that the request cannot be satisfied because one or more *uuid_key* values specified has previously been hidden or removed by the requester. This specifically applies to the tModelKey values passed.
- **E_invalidKeyPassed**: signifies that the request cannot be satisfied because one or more *uuid*_key values specified is not a valid key value, or that a hostingRedirector value references a bindingTemplate that itself contains a hostingRedirector value.
- **E_userMismatch**: signifies that one or more of the *uuid_key* values passed refers to data that is not controlled by the individual who is represented by the authentication token.
- **E_operatorMismatch**: signifies that one or more of the *uuid_key* values passed refers to data that is not controlled by the *Operator Site* that received the request for processing.
- **E_accountLimitExceeded**: signifies that user account limits have been exceeded.

save_business

The save_business message is used to save or update information about a complete businessEntity structure. This API has the broadest scope of all of the save_x API calls in the publisher API, and can be used to make sweeping changes to the published information for one or more businessEntity structures controlled by an individual.

Syntax:

```
<save_business  generic="1.0" xmlns="urn:uddi-org:api" >
    <authInfo/>
    <businessEntity/> [<businessEntity/>...] | <uploadRegister/> [<uploadRegister/>...]
</save_business>
```

Arguments:

Only one type of businessEntity or uploadRegister arguments may be passed in a given save_business message. Any number of businessEntity or uploadRegister values can be passed in a single save (up to an *Operator Site* imposed policy limit), but the two types of parameters should not be mixed.

- ***authInfo***: this required argument is an element that contains an authentication token. Authentication tokens are obtained using the get_authToken API call.
- ***businessEntity***: one or more complete businessEntity structures can be passed. These structures can be obtained in advance by using the get_businessDetail API call or by any other means.
- ***uploadRegister***: one or more resolvable HTTP URL addresses that each point to a single and valid businessEntity or businessEntityExt structure. This variant argument allows a registry to be updated to reflect the contents of an XML document that is URL addressable. The URL must return a pure XML document that only contains a businessEntity structure as its top-level element, and be accessible using the standard HTTP-GET protocol.

Behavior:

If any of the *uuid_key* values within in a businessEntity structure (e.g. any data with a key value regulated by a businessKey, serviceKey, bindingKey, or tModelKey) is passed with a blank value, this is a signal that the data that is so keyed is being inserted. This does not apply to structures that reference other keyed data, such as tModelKey references within bindingTemplate or keyedReference structures, since these are references.

To make this function perform an update to existing registered data, the keyed entities (businessEntity, businessService, bindingTemplate or tModel) should have *uuid_key* values that correspond to the registered data.

Data can be deleted with this function when registered information is different than the new information provided. One or more businessService and bindingTemplate structures that are found in the controlling *Operator Site* but are missing from the businessEntity information provided in or referenced by this call will be deleted from the registry after processing this call.

Data that is contained within one or more businessEntity can be rearranged with this function when data passed to this function redefines parent container relationships for other registered information. For instance, if a new businessEntity is saved with information about a businessService that is registered already as part of a separate businessEntity, this will result in the businessService being moved from its current container to the new businessEntity. This only applies if the same party controls the data referenced.

If the uploadRegister URL method is used to save data, the *Operator Site* is required to make sure that the URL used is included in the discoveryURLs collection within the businessEntity structure. If the URL passed to do the upload is not contained in this collection, it will be added automatically with a useType value set to the type of structure (either businessEntity or businessEntityExt) found in the file used to perform the upload.

If the file located by the uploadRegister URL value is an extended business entity (businessEntityExt) structure, only the businessEntity data found within that structure will be registered.

If the businessEntity method is used to save data (e.g not via an uploadRegister URL reference), then the *Operator Site* will create a URL that is specific to the *Operator Site* that can be used to get (via HTTP-GET) the businessEntity structure being registered. This information will be added to (if not present already) the discoveryURLs collection automatically with a useType value of "businessEntity".

Returns:

This API returns a businessDetail message containing the final results of the call that reflects the new registered information for the businessEntity information provided.

Caveats:

If any error occurs in processing this message, a dispositionReport structure will be returned to the caller in a SOAP Fault. The following error number information will be relevant:

- **E_authTokenExpired**: signifies that the authentication token value passed in the authInfo argument is no longer valid because the token has expired.
- **E_authTokenRequired**: signifies that the authentication token value passed in the authInfo argument is either missing or is not valid.
- **E_keyRetired**: signifies that the request cannot be satisfied because one or more *uuid_key* values specified has previously been hidden or removed by the requester. This specifically applies to the tModelKey values passed.
- **E_invalidKeyPassed**: signifies that the request cannot be satisfied because one or more *uuid*_key values specified is not a valid key value. This includes any tModelKey references that are unknown.
- **E_invalidURLPassed**: signifies that an error occurred with one of the uploadRegister URL values.
- **E_userMismatch**: signifies that one or more of the *uuid_key* values passed refers to data that is not controlled by the individual who is represented by the authentication token.
- **E_operatorMismatch**: signifies that one or more of the businessKey values passed refers to data that is not controlled by the *Operator Site* that received the request for processing.

- **E_invalidCategory (20000)**: signifies that the given keyValue did not correspond to a category within the taxonomy identified by a tModelKey value within one of the categoryBag elements provided.

- **E_categorizationNotAllowed (20100:** Restrictions have been placed by the taxonomy provider on the types of information that should be included at that location within a specific taxonomy. The validation routine chosen by the *Operator Site* has rejected this businessEntity for at least one specified category.

- **E_accountLimitExceeded**: signifies that user account limits have been exceeded.

save_service

The save_service message adds or updates one or more businessService structures.

Syntax:

```
<save_service  generic="1.0" xmlns="urn:uddi-org:api" >
    <authInfo/>
    <businessService/> [<businessService/>…]
</save_service>
```

Arguments:

- ***authInfo***: this required argument is an element that contains an authentication token. Authentication tokens are obtained using the get_authToken API call.
- ***businessService***: one or more complete businessService structures can be passed. These structures can be obtained in advance by using the get_serviceDetail API call or by any other means.

Behavior:

Each businessService structure passed must contain a businessKey value that corresponds to a registered businessEntity controlled by the same making the save_service request.. If the same businessService, or within these, bindingTemplate (determined by matching businessService or bindingKey value) is contained in more than one businessService argument, any relationship to the containing businessEntity will be determined by processing order – which is determined by first to last order of the information passed in the request. Using this message it is possible to move an existing bindingTemplate structure from one businessService structure to another, or move an existing businessService structure from one businessEntity to another by simply specifying a different parent businessEntity relationship. Changing a parent relationship in this way will cause two businessEntity structures to be affected.

Returns:

This API returns a serviceDetail message containing the final results of the call that reflects the newly registered information for the effected businessService structures.

Caveats:

If any error occurs in processing this message, a dispositionReport structure will be returned to the caller in a SOAP Fault. The following error number information will be relevant:

- **E_authTokenExpired:** signifies that the authentication token value passed in the authInfo argument is no longer valid because the token has expired.
- **E_authTokenRequired:** signifies that the authentication token value passed in the authInfo argument is either missing or is not valid.
- **E_keyRetired:** signifies that the request cannot be satisfied because one or more *uuid_key* values specified has previously been hidden or removed by the requester. This specifically applies to the tModelKey values passed.

* **E_invalidKeyPassed**: signifies that the request cannot be satisfied because one or more *uuid_*key values specified is not a valid key value. This includes any tModelKey references that are unknown.

* **E_userMismatch**: signifies that one or more of the *uuid_key* values passed refers to data that is not controlled by the individual who is represented by the authentication token.

* **E_operatorMismatch**: signifies that one or more of the *uuid_key* values passed refers to data that is not controlled by the *Operator Site* that received the request for processing.

* **E_invalidCategory (20000)**: signifies that a keyValue did not correspond to a category within the taxonomy identified by the tModelKey in the categoryBag data provided.

* **E_categorizationNotAllowed (20100:** The taxonomy validation routine chosen by the *Operator Site* has rejected the businessService data provided.

* **E_accountLimitExceeded**: signifies that user account limits have been exceeded.

save_tModel

The save_tModel message adds or updates one or more tModel structures.

Syntax:

```
<save_tModel  generic="1.0" xmlns="urn:uddi-org:api" >
    <authInfo/>
    <tModel/> [<tModel/>…] | <uploadRegister/> [<uploadRegister/>…]
</save_tModel>
```

Arguments:

* ***authInfo***: this required argument is an element that contains an authentication token. Authentication tokens are obtained using the get_authToken API call.

* ***tModel***: one or more complete tModel structures can be passed. If adding a new tModel, the tModelKey value should be passed as an empty element.

* ***uploadRegister***: one or more resolvable HTTP URL addresses that each point to a single and valid tModel structure. This variant argument allows a registry to be updated to reflect the contents of an XML document that is URL addressable. The URL must return a pure XML document that only contains a tModel structure as its top-level element, and be accessible using the standard HTTP-GET protocol.

Behavior:

If any of the *uuid_key* values within in a tModel structure (e.g. tModelKey) is passed with a blank value, this is a signal that the data is being inserted.

To make this function perform an update to existing registered data, the tModelKey values should have *uuid_key* values that correspond to the registered data. All tModelKey values that are non-blank are formatted as urn values (e.g. the characters "uuid:" precede all UUID values for tModelKey values)

If a tModelKey value is passed that corresponds to a tModel that was previously hidden via the delete_tModel message, the result will be the restoration of the tModel to full visibility (e.g. available for return in find_tModel results again).

Returns:

This API returns a tModelDetail message containing the final results of the call that reflects the new registered information for the effected tModel structures.

Caveats:

If any error occurs in processing this message, a dispositionReport structure will be returned to the caller in a SOAP Fault. The following error number information will be relevant:

- **E_authTokenExpired**: signifies that the authentication token value passed in the authInfo argument is no longer valid because the token has expired.
- **E_authTokenRequired**: signifies that the authentication token value passed in the authInfo argument is either missing or is not valid.
- **E_keyRetired**: signifies that the request cannot be satisfied because one or more *uuid_key* values specified has previously been hidden or removed by the requester. This specifically applies to the tModelKey values passed.
- **E_invalidKeyPassed**: signifies that the request cannot be satisfied because one or more *uuid*_key values specified is not a valid key value. This will occur if a *uuid_key* value is passed in a tModel that does not match with any known tModel key.
- **E_invalidURLPassed**: an error occurred with one of the uploadRegister URL values.
- **E_userMismatch**: signifies that one or more of the *uuid_key* values passed refers to data that is not controlled by the individual who is represented by the authentication token.
- **E_operatorMismatch**: signifies that one or more of the *uuid_key* values passed refers to data that is not controlled by the *Operator Site* that received the request for processing.
- **E_invalidCategory**: signifies that the given keyValue did not correspond to a category within the taxonomy identified by a tModelKey value within one of the categoryBag elements provided.
- **E_categorizationNotAllowed**: Restrictions have been placed by the taxonomy provider on the types of information that should be included at that location within a specific taxonomy. The validation routine chosen by the *Operator Site* has rejected this tModel for at least one specified category.
- **E_accountLimitExceeded**: signifies that user account limits have been exceeded.

Appendix A: Error code reference

Error Codes

The following list of error codes can be returned in the errno values within a dispositionReport response to the API calls defined in this programmer's reference. The descriptions in this section are general and when used with the specific return information defined in the individual API call descriptions are useful for determining the reason for failures.

- **E_authTokenExpired**: (10110) signifies that the authentication token information has timed out.
- **E_authTokenRequired**: (10120) signifies that an invalid authentication token was passed to an API call that requires authentication.
- **E_accountLimitExceeded**: (10160) signifies that a save request exceeded the quantity limits for a given structure type. See "Structure Limits" in Appendix D for details.
- **E_busy**: (10400) signifies that the request cannot be processed at the current time.
- **E_categorizationNotAllowed**: (20100) Restrictions have been placed by the on the types of information that can categorized within a specific taxonomy. The data provided does not conform to the restrictions placed on the category used. Used with categorization only.
- **E_fatalError**: (10500) signifies that a serious technical error has occurred while processing the request.
- **E_invalidKeyPassed**: (10210) signifies that the uuid_key value passed did not match with any known key values. The details on the invalid key will be included in the dispositionReport structure.
- **E_invalidCategory** (20000): signifies that the given keyValue did not correspond to a category within the taxonomy identified by the tModelKey. Used with categorization only.
- **E_invalidURLPassed**: (10220) signifies that an error occurred during processing of a save function involving accessing data from a remote URL. The details of the HTTP Get report will be included in the dispositionReport structure.
- **E_keyRetired**: (10310) signifies that a *uuid_key* value passed has been removed from the registry. While the key was once valid as an accessor, and is still possibly valid, the publisher has removed the information referenced by the *uuid_key* passed.
- **E_languageError**: (10060) signifies that an error was detected while processing elements that were annotated with xml:lang qualifiers. Presently, only the description element supports xml:lang qualifiacations.
- **E_nameTooLong**: (10020) signifies that the partial name value passed exceeds the maximum name length designated by the policy of an implementation or *Operator Site*.
- **E_operatorMismatch**: (10130) signifies that an attempt was made to use the publishing API to change data that is mastered at another *Operator Site*. This error is only relevant to the public *Operator Sites* and does not apply to other UDDI compatible registries.

- **E_success**: (0) Signifies no failure occurred. This return code is used with the dispositionReport for reporting results from requests with no natural response document.
- **E_tooManyOptions**: (10030) signifies that incompatible arguments were passed.
- **E_unrecognizedVersion**: (10040) signifies that the value of the *generic* attribute passed is unsupported by the *Operator Instance* being queried.
- **E_unknownUser**: (10150) signifies that the user ID and password pair passed in a get_authToken message is not known to the *Operator Site* or is not valid.
- **E_unsupported**: (10050) signifies that the implementer does not support a feature or API.
- **E_userMismatch**: (10140) signifies that an attempt was made to use the publishing API to change data that is controlled by another party. In certain cases, E_operatorMismatch takes precedence in reporting an error.

dispositionReport overview

Errors that are not reported by way of SOAP Faults are reported using the dispositionReport structure. This structure can be used to signal success for asynchronous requests as well.

Success Reporting with the dispositionReport structure

The general form of a success report is:

```
<?xml version="1.0" encoding="UTF-8" ?>

<Envelope xmlns="http://schemas.xmlsoaporg.org/soap/envelope/">
<Body>
   <dispositionReport generic="1.0" operator="OperatorUrl"
        xmlns="urn:uddi-org:api" >
      <result errno="0" >
         <errInfo errCode="E_success" />
      </result>
   </dispositionReport>
</Body>
</Envelope>
```

Error reporting with the dispositionReport structure

All application errors are communicated via the use of the SOAP FAULT structure. The general form of an error report is:

```
<?xml version="1.0" encoding="UTF-8" ?>

<Envelope xmlns="http://schemas.xmlsoaporg.org/soap/envelope/">
<Body>
   <Fault>
        <faultcode>Client</faultcode>
        <faultstring>Client Error</faultstring>
```

```
            <detail>
              <dispositionReport generic="1.0" operator="OperatorUrl"
                  xmlns="urn:uddi-org:api" >
                <result errno="10050" >
                  <errInfo errCode="E_notSupported">
                    The findQualifier value passed is unrecognized.
                  </errInfo>
                </result>
              </dispositionReport>
          </detail>
        </Fault>
      </Body>
    </Envelope>
```

Multiple *result* elements may be present within the dispositionReport structure, and can be used to provide very detailed error reports for multiple error conditions. The number of *result* elements returned within a disposition report is implementation specific. In general it is permissible to return an error response as soon as the first error in a request is detected.

Appendix B: SOAP usage details

This appendix covers the SOAP specific conventions and requirements for *Operator Sites*.

Support for SOAPAction

In version 1, the SOAPAction HTTP Header is required. The value passed in this HTTP Header must be an empty string that is surrounded by double quotes. Example:

```
POST /get_BindingDetail HTTP/1.1
Host: www.someOperator.com
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn
SOAPAction: ""
```

Support for SOAP Actor

In version 1 of the UDDI specification, the SOAP Actor feature is not supported. *Operator Sites* will reject any request that arrives with a SOAP Actor attribute.

Support for SOAP encoding

In version 1 of the UDDI specification, the SOAP encoding feature (section 5) is not supported. *Operator Sites* will reject any request that arrives with a SOAP encoding attribute.

Support for SOAP Fault

SOAP Fault applies when unknown API references invoked, etc. Applegate specific errors will be handled via the dispositionReport API within SOAP Fault structures (see appendix A). The following SOAP fault codes are used:

- **VersionMismatch**: An invalid namespace reference for the SOAP envelope element was passed. The valid namespace value is "http://www.xmlsoap.org/soap/envelope/".

- **MustUnderstand**: A SOAP header element was passed to an *Operator Site*. *Operator Sites* do not support any SOAP headers, and will return this error whenever a SOAP request is received that contains any Headers element.

- **Client**: A message was incorrectly formed or did not contain enough information to perform more exhaustive error reporting.

Support for SOAP Headers

In version 1 of the UDDI specification, SOAP Headers are not supported. *Operator Sites* are permitted to ignore any headers received. SOAP headers that have the must_understand attribute set to true will be rejected with a SOAP fault - MustUnderstand.

Document encoding conventions – default namespace support

Operator Sites are required to support the use of the default namespaces in SOAP request and response documents as shown in the following HTTP example:

```
POST /get_bindingDetail HTTP/1.1
Host: www.someoperator.org
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn
SOAPAction: ""

<?xml version="1.0" encoding="UTF-8" ?>
<Envelope xmlns="http://schemas.xmlsoap.org/soap/envelope/">
      <Body>
         <get_bindingDetail generic="1.0"
              xmlns="urn:uddi-org:api">
  ...
```

UTF-8 to Unicode: SOAP listener behavior

The decision to use the UTF-8 encoding in all requests simplified the number of encoding variations that need to be handled within the XML interchanges used in this API specification. However, byte ordering and conversion issues can still arise. This section describes the behavior of the SOAP listeners that run at *Operator Sites* in regards to the way they convert data received into XML encoded in Unicode.

UTF-8 allows data to be transmitted with an optional three-position byte order mark (BOM) preceding the XML data. This BOM does not contain information that is useful for decoding the contents, but tells the receiving program the order that bytes within double-byte pairs occur within the data. Further analysis can then be performed to determine whether the XML received contains ASCII or Unicode characters. The BOM is not required to perform this analysis however, and it is safe for *Operator Sites* to remove the BOM prior to processing messages received.

Operator Sites must be prepared to accept messages that contain Byte Order Marks, but the BOM is not required to process SOAP messages successfully.

Data returned by all of the messages defined in this specification will not contain a BOM, and will be encoded as UTF-8 XML data.

Appendix C: XML Usage Details

This appendix explains the specifics of XML conventions employed across all UDDI *Operator Sites*. Implementations that desire to remain compliant with the behaviors of *Operator Sites* should follow these same conventions.

Use of multiple languages in the description elements.

Many of the messages defined in this programmers interface specification contain an element named *description*. Multiple descriptions are allowed to accommodate multiple language descriptions. These description elements are also permitted to be sent without an xml:lang attribute qualifier.

Only one description element is allowed to be passed to a save_xx API call without an xml:lang attribute qualifier. Elements passed in this way will be assigned the default language code of the registering party. This default language code is established at the time that publishing credentials are established with an individual *Operator Site*.

If more than one description element is sent in a document being stored, only the first description element in a particular structure may be sent without a xml:lang attribute qualifier. All subsequent description peers must contain an xml:lang qualifier.

Valid Language Codes

The valid values for language codes are to be specified as ISO language codes. Values for these codes and translation to other common language code sets can be found at:

http://www.unicode.org/unicode/onlinedat/languages.html

Only one description element is allowed for each language code used at any given container level.

Default Language Codes

A default ISO language code will be determined for a publisher at the time that a party establishes permissions to publish at a given *Operator Site* or implementation. This default language code will be applied to any description values that are provided with no language code.

On data returned via the SOAP interface, all descriptions will contain xml:lang qualifications.

ISSUE for Validation: XML namespace declaration

For use with the xml:lang language qualifiers, documents that contain this attribute will declare the xml namespace as:

xmlns:xml="http://www.w3.org/1999/XMLSchema"

This will occur at the top level of the instance document (in the Envelope element)

Support for XML Encoding

All messages to and from the Operator Site shall be encoded using the UTF-8 encoding, and all such messages shall have the 'encoding="UTF-8"' attribute on the initial line. Other encoding name variants, such as UTF8, UTF_8, etc. shall not be used. Therefore, to be explicit, the initial line shall be:

```
<?xml version="1.0" encoding="UTF-8" ?>
```

Appendix D: Security model in the publishers API

The Publishers API describes the messages that are used to control the content contained within an *Operator Site*, and can be used by compliant non-operator implementations that adhere to the behaviors described in this programmers reference specification.

Achieving wire level privacy: All methods are secured via SSL

All calls made to *Operator Sites* that use the messages defined in the publishers API will be transported using SSL encryption. *Operator Sites* will each provide a service description that exposes a bindingTemplate that makes use of HTTPS and SSL to secure the transmission of data.

Authentication

Each of the calls in the publishers API that change information at a given *Operator Site* requires the use of an opaque authentication token. These tokens are generated by or provided by each *Operator Site* independently, and are passed from the caller to the *Operator Site* in the element named *authInfo*.

These tokens are meaningful only to the *Operator Site* that provided them and are to be used according to the published policies of a given *Operator Site*.

Each party who has been granted publication access to a given *Operator Site* will be provided a token by the site. Obtaining this token is *Operator Site* specific.

Establishing credentials

Before any party can publish data within an *Operator Site*, credentials and permission to publish must be established with the individual operator. Generally, you will only need to interact with one *Operator Site* because all data published at any *Operator Site* is replicated automatically to all other *Operator Sites*. Establishing publishing credentials involves providing some verifiable identification information, contact information and establishing security credentials with the individual *Operator Site*. The specifics of these establishing credentials is *Operator Site* dependant, and all valid *Operator Sites* will provide a Web-based user interface via which to establish an identity and secure permissions to publish data.

Authentication tokens are not portable

Every registry implementation that adheres to these specifications will establish their own mechanism for token generation and authentication. The only requirement placed on token generation for use with the publishers API is that the tokens themselves must be valid string text that can be placed within the authInfo XML element. Given that binary to string translations are well understood and in common use, this requirement will not introduce hardships.

Authentication tokens are not required to be valid except at the *Operator Site* or implementation from which they originated. These tokens need only have meaning at a single *Operator Site* or implementation, and will not be expected to work across sites.

Generating Authentication Tokens

Many implementations are expected to require a login step. The get_authToken message is provided to accommodate those implementations that desire a login step. Security schemes that are based on the convention of exchanging User ID and password credentials fall into this category. For implementations that desire this kind of security, the get_authToken API is provided as an optional means for generating a temporary authentication token.

Certificate based authentication and similar security mechanisms do not require this additional step of "logging in" and can directly pass compatible authentication token information (such as a certificate value) within the authInfo element provided on each of the publishers API messages. If certificate based authentication or similar security is employed by the choice of a given *Operator Site*, the use of the get_authToken and discard_authToken messages is optional.

Per-account space limits

Operator Sites may impose limits on the amount of data that can be published by a given user. The initial limits for a new user are:

- businessEntity: 1 per user account
- businessService: 4 per businessEntity
- bindingTemplate: 2 per businessService
- tModel: 10 per user account

Individual user accounts can negotiate per-account limits with the *Operator Site*.

Appendix E: Search Qualifiers

The inquiry API functions *find_binding*, *find_business*, *find_service*, and *find_tModel* each will accept an optional element named findQualifiers. This element argument is provided as a means to allow the caller to override default search behaviors.

General form of search qualifiers

The general form of the findQualifiers structure is:

```
<findQualifiers>
   <findQualifier>fixedQualifierValue</findQualifier>
   [<findQualifier>fixedQualifierValue</findQualifier> …]
</findQualifiers>
```

Search Qualifiers enumerated

The value passed in each findQualifier element represents the behavior change desired by the caller. These values must come from the following list of qualifiers:

- **exactNameMatch**: signifies that leftmost name match behavior should be overridden. When this behavior is specified, only entries that exactly match the entry passed in the name argument will be returned.

- **caseSentiveMatch**: signifies that the default case-insensitive behavior of a name match should be overridden. When this behavior is specified, case is relevant in the search results and only entries that match the case of the value passed in the name argument will be returned.

- **sortByNameAsc**: signifies that the result returned by a *find_x* or *get_x* inquiry call should be sorted on the name field in ascending alphabetic sort order. This sort is applied prior to any truncation of result sets. Only applicable on queries that return a *name* element in the topmost detail level of the result set. If no conflicting sort qualifier is specied, this is the default sort order for inquiries that return *name* values at this topmost detail level.

- **sortByNameDesc**: signifies that the result returned by a *find_x* or *get_x* inquiry call should be sorted on the name field in descending alphabetic sort order. This sort is applied prior to any truncation of result sets. Only applicable on queries that return a *name* element in the topmost detail level of the result set. This is the reverse of the default sort order for this kind of result.

- **sortByDateAsc**: signifies that that the result returned by a *find_x* or *get_x* inquiry call should be sorted based on the date last updated in ascending chronological sort order (earliest returns first). If no conflicting sort qualifier is specified, this is the default sort order for all result sets. Sort qualifiers involving date are secondary in precedence to the sortByName qualifiers. This causes sortByName elements to be sorted within name by date, oldest to newest.

- **sortByDateDesc**: (default) signifies that the result returned by a *find_x* or *get_x* inquiry call should be sorted based on the date last updated in descending chronological sort order (most recent change returns first). Sort qualifiers involving date are secondary in precedence to the sortByName qualifiers. This causes sortByName elements to be sorted within name by date, oldest to newest.

At this time, these are the only qualifiers defined. *Operator Sites* may define more search qualifier values than these – but all *Operator Sites* and fully compatible software must support these qualifiers and behaviors.

Search Qualifier Precedence

Precedence of search qualifiers, when combined is as follows:

1. **exactNameMatch**, **caseSensitiveMatch**: These can be combined but are equal in precedence.

2. **sortByNameAsc**, **sortByNameDesc**: These are mutually exclusive, but equal in precedence.

3. **sortByDateAsc**, **sortByDateDesc**: These are mutually exclusive, but equal in precedence.

The precedence order is used to determine the proper ordering of results when multiple search qualifiers are combined.

Locale Details

The US English (EN_US) locale shall be used whenever a string comparison or alphabetic sort is specified. This applies to sortByNameAsc, sortByNameDesc, exactNameMatch.

Appendix F: Response message reference

Here we explain each of the response messages. These are technically defined in the UDDI API schema:

- **authToken**: This structure is return by the optional get_authToken message to return authentication information. The value returned is used in subsequent calls that require an authInfo value.
- **bindingDetail**: This structure is the technical information required to make a method call to an advertised web service. It is returned in response to the get_bindingDetail message.
- **businessDetail**: This structure contains full details for zero or more businessEntity structures. It is returned in response to a get_businessDetail message, and optionally in response to the save_business message.
- **businessDetailExt**: This structure allows UDDI compatible registries to define and share extended information about a businessEntity. *Operator Sites* support this message but return no additional data. This structure contains zero or more businessEntityExt structures. It is returned in response to a get_businessDetailExt message.
- **businessList**: This structure contains abbreviated information about registered businessEntity information. This message contains zero or more businessInfo structures. It is returned in response to a find_business message.
- **dispositionReport**: This structure is used to report the outcome of message processing and to report errors discovered during processing. This message contains one or more result structures. A special case – success – contains only one result structure with the special errno attribute value of E_success (0).
- **registeredInfo**: This structure contains abbreviated information about all registered businessEntity and tModel information that are controlled by the party specified in the request. This message contains one or more businessInfo structures and zero or more tModelInfo structures. It is returned in response to a get_registeredInfo message.
- **serviceDetail**: This structure contains full details for zero or more businessService structures. It is returned in response to a get_serviceDetail message, and optionally in response to the save_binding and save_service messages.
- **serviceList**: This structure contains abbreviated information about registered businessService information. This message contains zero or more serviceInfo structures. It is returned in response to a find_service message.
- **tModelDetail**: This structure contains full details for zero or more tModel structures. It is returned in response to a get_tModelDetail message, and optionally in response to the save_tModel message.
- **tModelList**: This structure contains abbreviated information about registered tModel information. This message contains zero or more tModelInfo structures. It is returned in response to a find_tModel message.

Appendix G: redirection via hostingRedirector element

One of the main benefits of using a public *Operator* Site instance of an UDDI registry is to provide a single point of reference for determining the correct location to send a business service request to a remote web service. In general, the controller of a particular instance of bindingTemplate structure can be assured that by keeping the registered copy pointing to the proper server or invocation address, special conditions such as disaster recovery to a secondary site can be handled with a minimum of service disruption for customers or partners. The same holds true for those who choose to use a registry that is compatible with the UDDI API, but to a lesser degree.

In many cases, the API specified in the get_bindingDetail message is straightforward. Once a business or application knows of a service that needs to be invoked, the bindingTemplate information for this service can be cached until needed. In the event that the cached information fails at the time the partner web service is actually invoked (e.g. the accessPoint information in the cached bindingTemplate structure is used to invoke a remote partner service), the application can use the bindingKey in the cached information to get a fresh copy of the bindingTemplate information. This cached approach serves to prevent needless round trips to the registry.

Special situations requiring the hostingRedirector

Two special needs arise that cannot be directly supported by the *accessPoint* information in a bindingTemplate. These are:

- **Third Party Hosting of Technical Web Services**: A business chooses to expose a service that is actually hosted at a remote or third party site. Application Service Providers and Network Market Makers are common examples of this situation. In this situation, it is the actual third party that needs to control the actual value of the binding information.
- **Use specific access control to binding location**: In other situations, such as situation specific redirection based on the identity of the caller, or even time-of-day routing, it is necessary to provide the actual contact point information for the remote service in a more dynamic way than the cached accessPoint data would support.

For these cases, the bindingTemplate structures contain an alternative data element called *hostingRedirector*. The presence of a hostingRedirector element is mutually exclusive with the accessPoint information. This makes it possible to tell which method of gaining the actual bindingTemplate information that contains the accessPoint data to use.

Using the hostingRedirector data

When a bindingTemplate returned by UDDI registry contains a hostingRedirector element, the programmer uses this information to locate the actual bindingTemplate for the hosted service. The content of the hostingRedirector element is a bindingKey reference that refers to another bindingTemplate that contains the address of a redirector service that will respond to a get_bindingDetail message. The argument that gets passed to this message is the original bindingTemplate *uuid_key* value for the redirected service. The bindingTemplate returned by this redirected call must have a accessPoint element in it – this being the actual binding information for the redirected web service request

Stepwise overview

1. A business registers a bindingTemplate ***A*** for a remotely hosted or redirected business service *S*. This bindingTemplate contains a bindingKey value ***Q*** that references a second

bindingTemplate ***B***. The bindingTemplate ***B*** is typically controlled by the organization that hosts the redirection service. This bindingTemplate ***B*** contains an accessPoint element that points to the actual hostingRedirector service ***R***.

2. A program that wants to call the service ***S*** that is registered in step one gets the binding information for the advertised service. This bindingTemplate information contains a hostingRedirector element with the bindingKey ***K*** for the bindingTemplate ***B***.

3. The programmer takes the bindingKey ***K*** for and issues a get_bindingDetail message against the UDDI registry that the original bindingTemplate ***A*** came from. This returns the data for bindingTemplate ***B***. The programmer now has the address of the service that implements the redirection. This information is in the accessPoint element found in bindingTemplate ***B***. This service, to be compliant, knows how to respond to a get_bindingDetail message.

4. Using the original binding key ***Q*** and issues a get_bindingDetails to the redirector service ***R***. This service is responsible for returning the actual binding information for the redirected business service ***S*** or returning an error. The programmer has the choice of caching this bindingTemplate if desired.

Using this algorithm, an organization that hosts services for other businesses to use can control the information that is used to actually access the hosted service. This not only provides this hosting organization with the ability to manage situations such as disaster recovery locations, but also lets them specify the actual URL that is used to make a call to the actual business service. This URL can be keyed specifically to the caller, or can be a general location for the hosted or redirected service.

In any case, the original caller is able to find the technical web service (bindingTemplate) advertised within the actual business partner's data without having to know that any redirection occurred.

Appendix H: Details on the validate_categorization call

Whenever save_business, save_service or save_tModel are called, all contents of any included *categoryBag* information may be checked to see that it is properly coded to match existing categories. The reason given for this is to maximize consistency across category based searches.

Operator specific policy allows interpretation of various error returns to result in non-specified behavior. Consult the operations policy of the operator at which you register data to understand the specific behaviors of any validation performed.

validate_categorization

The validate_categorization service performs two functions. It is used to verify that a specific category (keyValue) exists within the given taxonomy. The service also optionally restricts the entities that may be classified within the category.

Syntax:

```
<validate_categorization  generic="1.0" xmlns="name_qualifier" >
   <tModelKey/>
   <keyValue/>
   [ <businessEntity/> | <businessService/> | <tModel/> ]
</validate_categorization>
```

Arguments:

The optional businessEntity, businessService or tModel parameters are mutually exclusive.

- ***tModelKey***: The identifier of a registered tModel that is used as a namespace qualifier that implies a specific taxonomy.
- **keyValue:** The *category identifier*[3] of the category within the identified taxonomy.
- **businessEntity:** (optional) The businessEntity structure being categorized
- **businessService:** (optional) The business service structure being categorized.
- **tModel:** (optional) The tModel structure being categorized.

Behavior:

To validate categorizations of registry entries, it is sufficient to verify the category identified by the keyValue parameter exists within the taxonomy identified by the tModelKey.

Optionally, a businessEntity, businessService, or tModel may be passed with the required values. This additional information may be used by the taxonomy validation service to verify that the entity is properly classified within this taxonomy category.

[3] A *category identifier* is the specific coded value that designates a category within taxonomy. An example would be NAICS code 247 with the appropriate category identifier being "247".

Returns:

Upon successful completion, a dispositionReport is returned with a single success indicator.

Caveats:

If any error occurs in processing this message, a dispositionReport structure will be returned to the caller. The following error number information will be relevant:

- **E_invalidKeyPassed**: the tModelKey value passed didn't match any known tModel.
- **E_invalidCategory**: one of the keyValue values supplied did not correspond to a category within the taxonomy identified by the tModelKey.
- **E_categorizationNotAllowed**: The optional businessEntity, businessService, or tModel provided does not conform to restrictions placed on the given category by the taxonomy publisher.

Appendix I: Utility tModels and Conventions

In order to facilitate consistency in Service Description (tModel) registration, and provide a framework for their basic organization within the UDDI registry, a set of conventions has been established. This section describes the conventions for registration of Service Descriptions, as well as a set of Utility tModels that facilitate registration of common information and the services provided by the UDDI registry itself.

UDDI Type Taxonomy

The UDDI specifications provide a great deal of flexibility in terms of the types of information that may be registered. A type taxonomy has been established to assist in general categorization of the types of information registered. In this release, the type taxonomy has been developed for the categorization of Service Descriptions, or tModels. Business or Service types may be incorporated into this taxonomy at a later date.

The approach to categorization of tModels within the UDDI Type Taxonomy is consistent with that used for each of the other taxonomies. The categorization information for each tModel is added to the `<categoryBag>` elements in a **save_tModel** message. A `<keyedReference>` element is added to the category bag to indicate the type of tModel that is being registered.

The values used for keyed references are defined in the UDDI Type Taxonomy shown in the tModel description table below.

tModel Name:	`uddi-org:types`
tModel Description:	UDDI Type Taxonomy
tModel UUID:	`uuid:C1ACF26D-9672-4404-9D70-39B756E62AB4`

Taxonomy Values

The table below describes the UDDI types taxonomy. As the structure is hierarchical, the ParentID column indicates the parent-child relationships. The tModel key is the root of the structure. Categorization is allowed at all levels of the taxonomy, with the exception of the root key.

ID	ParentID	Allowed	Description
tModel	tModel	no	These types are used for tModels
identifier	tModel	yes	Unique identifier
namespace	tModel	yes	Namespace
categorization	tModel	yes	Categorization (taxonomy)
specification	tModel	yes	Specification for a Web Service
xmlSpec	specification	yes	Specification for a Web Service using XML messages

soapSpec	xmlSpec	yes	Specification for interaction with a Web Service using SOAP messages
wsdlSpec	specification	yes	Specification for a Web Service described in WSDL
protocol	tModel	yes	Protocol
transport	protocol	yes	Wire/transport protocol
signatureComponent	tModel	yes	Signature component

tModel: The UDDI type taxonomy is structured to allow for categorization of registry entries other than tModels. This key is the root of the branch of the taxonomy that is intended for use in categorization of tModels within the UDDI registry. Categorization is not allowed with this key.

identifier: An identifier tModel represents a specific set of values used to uniquely identify information. For example, a Dun & Bradstreet D-U-N-S® Number uniquely identifies companies globally. The D-U-N-S® Number taxonomy is an identifier taxonomy.

namespace: A namespace tModel represents a scoping constraint or domain for a set of information. In contrast to an identifier, a namespace does not have a predefined set of values within the domain, but acts to avoid collisions. It is similar to the namespace functionality used for XML.

categorization: A categorization tModel is used for information taxonomies within the UDDI registry. NAICS and UNSPSC are examples of categorization tModels.

specification: A specification tModel is used for tModels that define interactions with a Web Service. These interactions typically include the definition of the set of requests and responses or other types of interaction that are prescribed by the service. tModels describing XML, COM, Corba, or any other services are specification tModels.

xmlSpec: An xmlSpec tModel is a refinement of the specification tModel type. It is used to indicate that the interaction with the service is via XML. The UDDI API tModels are xmlSpec tModels.

soapSpec: Further refining the xmlSpec tModel type, a soapSpec is used to indicate that the interaction with the service is via SOAP. The UDDI API tModels are soapSpe tModels, in addition to xmlSpec tModels.

wsdlSpec: A tModel for a Web Service described using WSDL is categorized as a wsdlSpec.

protocol: A tModel describing a protocol of any sort.

transport: A transport tModel is a specific type of protocol. HTTP, FTP, and SMTP are types of transport tModels.

signatureComponent: A signature component is used to for cases where a single tModel can not represent a complete specification for a Web Service. This is the case for specifications like RosettaNet, where implementation requires the composition of three tModels to be complete - a general tModel indicating RNIF, one for the specific PIP, and one for the error handling services.

Each of these tModels would be of type signature component, in addition to any others as appropriate.

UDDI Registry tModels

The UDDI registry defines a number of tModels to define its core services. Each of the core tModels are listed in this section.

tModel Name:	`uddi-org:inquiry`
tModel Description:	UDDI Inquiry API - Core Specification
tModel UUID:	`uuid:4CD7E4BC-648B-426D-9936-443EAAC8AE23`
Categorization:	`specification, xmlSpec, soapSpec`

This tModel defines the inquiry API calls for interacting with the UDDI registry.

tModel Name:	`uddi-org:publication`
tModel Description:	UDDI Publication API - Core Specification
tModel UUID:	`uuid:64C756D1-3374-4E00-AE83-EE12E38FAE63`
Categorization:	`specification, xmlSpec, soapSpec`

This tModel defines the publication API calls for interacting with the UDDI registry.

tModel Name:	`uddi-org:taxonomy`
tModel Description:	UDDI Taxonomy API
tModel UUID:	`uuid:3FB66FB7-5FC3-462F-A351-C140D9BD8304`
Categorization:	`specification, xmlSpec, soapSpec`

This tModel defines the taxonomy maintenance API calls for interacting with the UDDI registry.

UDDI Core tModels - Taxonomies

An additional set of tModels has been established to assist in categorization within industry taxonomies. There tModels are described below.

tModel Name:	`ntis-gov:naics:1997`
tModel Description:	Business Taxonomy: NAICS (1997 Release)
tModel UUID:	`uuid:C0B9FE13-179F-413D-8A5B-5004DB8E5BB2`

Categorization:	`categorization`

This tModel defines the NAICS industry taxonomy.

tModel Name:	`unspsc-org:unspsc:3-1`
tModel Description:	Product Taxonomy: UNSPSC (Version 3.1)
tModel UUID:	`uuid:DB77450D-9FA8-45D4-A7BC-04411D14E384`
Categorization:	`categorization`

This tModel defines the UNSPSC product taxonomy.

tModel Name:	`uddi-org:misc-taxonomy`
tModel Description:	Other Taxonomy
tModel UUID:	`uuid:A035A07C-F362-44dd-8F95-E2B134BF43B4`
Categorization:	`categorization`

This tModel defines an unidentified taxonomy.

UDDI Core tModels - Other

Additional tModels are defined to help register within leading industry encoding schemes and standard protocols. This list is expected to be expanded as appropriate as the UDDI business registry expands.

tModel Name:	`dnb-com:D-U-N-S`
tModel Description:	Dun & Bradstreet D-U-N-S® Number
tModel UUID:	`uuid:8609C81E-EE1F-4D5A-B202-3EB13AD01823`
Categorization:	`identifier`

This tModel is used for the Dun & Bradstreet D-U-N-S® Number identifier. Note that this tModel is initially registered as part of the UDDI core tModels. Once the registry is in production, management of this tModel is expected to be transferred to the Dun & Bradstreet publisher account. For more information, see http://www.dnb.com.

tModel Name:	`thomasregister-com:supplierID`
tModel Description:	Thomas Registry Suppliers
tModel UUID:	`uuid:B1B1BAF5-2329-43E6-AE13-BA8E97195039`
Categorization:	`identifier`

This tModel is used for the Thomas Register supplier identifier codes. Note that this tModel is initially registered as part of the UDDI core tModels. Once the registry is in production, custody of this tModel is expected to be transferred to the Thomas Register publisher account. For more information, see http://www.thomasregister.com.

tModel Name:	`uddi-org:smtp`
tModel Description:	E-mail based web service
tModel UUID:	`uuid:93335D49-3EFB-48A0-ACEA-EA102B60DDC6`
Categorization:	`transport`

This tModel is used to describe a web service that is invoked through SMTP email transmissions. These transmissions may be either between people or applications.

tModel Name:	`uddi-org:fax`
tModel Description:	Fax based web service
tModel UUID:	`uuid:1A2B00BE-6E2C-42F5-875B-56F32686E0E7`
Categorization:	`protocol`

This tModel is used to describe a web service that is invoked through fax transmissions. These transmissions may be either between people or applications.

tModel Name:	`uddi-org:ftp`
tModel Description:	File transfer protocol (ftp) based web service
tModel UUID:	`uuid:5FCF5CD0-629A-4C50-8B16-F94E9CF2A674`
Categorization:	`transport`

This tModel is used to describe a web service that is invoked through file transfers via the ftp protocol.

tModel Name:	`uddi-org:telephone`
tModel Description:	Telephone based web service
tModel UUID:	`uuid:38E12427-5536-4260-A6F9-B5B530E63A07`
Categorization:	`specification`

This tModel is used to describe a web service that is invoked through a telephone call and interaction by voice and/or touch-tone.

tModel Name:	`uddi-org:http`
tModel Description:	An http or web browser based web service
tModel UUID:	`uuid:68DE9E80-AD09-469D-8A37-088422BFBC36`
Categorization:	`transport`

This tModel is used to describe a web service that is invoked through a web browser and/or the http protocol.

Registering tModels within the Type Taxonomy

When a new tModel is registered within UDDI, its type can be classified within the framework of the UDDI Type Taxonomy. This classification provides additional hints to applications for what type of tModel is being registered. For each appropriate classification, and keyed reference is added to the category bag element for the tModel.

As an example, the Dun & Bradstreet D-U-N-S® Number is a type of identifier for an organization. Within the UDDI type taxonomy, the `dnb-com:D-U-N-S` tModel is classified as type identifier.

The categoryBag element of the tModel registered would be as follows :

```
<categoryBag>
 <keyedReference
   tModelKey = "uuid:C1ACF26D-9672-4404-9D70-39B756E62AB4"
   keyValue = "identifier"
   keyName = "tModel is a unique identifier">
</categoryBag>
```

tModelKey: This is the GUID for the UDDI Types taxonomy. It is required.

keyValue: This is the identifier for the categorization within the UDDI Types taxonomy. It is required.

keyName: This is the description of the identifier within the UDDI Types taxonomy. It is not required as a part of the registration, but simply provides additional information about the key selected.

References

This section contains URL pointers to various specifications and other documents that are pertinent in understanding this specification.

W3C specifications, notes and drafts

- XML 1.0
- XML Schema
- XML namespaces
- SOAP 1.1

UDDI specifications, white-papers and schemas

- UDDI API schema
- UDDI overview
- UDDI technical overview
- UDDI Operators specification
- UDDI.org
- UDDI XML structure reference

Change History

V1.00 30 September 2000 (Christopher Kurt). Added Appendix I, and updated table of contents as appropriate. Revised document date for final publication.

V1.01 27 March 2001 (Tom Glover). Corrected typographical errors.

uddi.org
Universal Description, Discovery and Integration

Exhibit D

UDDI Version 2.03 Data Structure Reference
UDDI Published Specification, 19 July 2002

This version:

http://uddi.org/pubs/DataStructure-V2.03-Published-20020719.pdf

Latest version:

http://uddi.org/pubs/DataStructure_v2.pdf

Editors (alphabetically):

David Ehnebuske, IBM
Dan Rogers, Microsoft
Claus von Riegen, SAP

Contributors (alphabetically):

Tom Bellwood, IBM
Andy Harris, i2 Technologies
Denise Ho, Ariba
Yin-Leng Husband, Compaq
Alan Karp, HP
Keisuke Kibakura, Fujitsu
Jeff Lancelle, Verisign
Sam Lee, Oracle
Sean MacRoibeaird, Sun
Barbara McKee, IBM
Tammy Nordan, Compaq
Dan Rogers, Microsoft
Christine Tomlinson, Sun
Cafer Tosun, SAP

Contents

1 Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119.

2 Introduction

The programmatic interface provided for interacting with systems that follow the Universal Description Discovery & Integration (UDDI) specifications make use of Extensible Markup Language (XML) and a related technology called Simple Object Access Protocol (SOAP), which is a specification for using XML in simple message based exchanges.

The UDDI Version 2.0 API Specification defines approximately 40 SOAP messages that are used to perform inquiry and publishing functions against any UDDI compliant service registry. This document outlines the details of each of the XML structures associated with these messages.

2.1 Service Discovery

The purpose of UDDI compliant registries is to provide a service discovery platform on the World Wide Web. Service discovery is related to being able to advertise and locate information about different technical interfaces exposed by different parties. Services are interesting when you can discover them, determine their purpose, and then have software that is equipped for using a particular type of Web service complete a connection and derive benefit from a service.

A UDDI compliant registry provides an information framework for describing services exposed by any entity or business. In order to promote cross platform service description that is suitable to a "black-box[1]" Web environment, this description is rendered in cross-platform XML.

2.1.1 Five data types

The information that makes up a registration consists of five data structure types. This division by information type provides simple partitions to assist in the rapid location and understanding of the different information that makes up a registration.

The five core types are shown in figure 1.

These five types make up the complete amount of information provided within the UDDI service description framework. Each of these XML structures contains a number of data fields[2] that serve either a business or technical descriptive purpose. Explaining each of these structures and the meaning and placement of each field is the primary purpose of this document.

These structures are defined in the UDDI Version 2.0 API schema. The schema defines approximately 25 requests and 15 responses, each of which contain these structures, references to these structures, or summary versions of these structures. In this document we first explain the core structures, and then provide descriptions of the individual structures used for the request/response XML SOAP interface.

---

[1] The term "black box" in this context implies that the descriptive information found in a UDDI compliant registry is provided in a neutral format that allows any kind of service, without regard to a given services platform requirements or technology requirements. UDDI provides a framework for describing any kind of service, and allows storage of as much detail about a service and its implementation as desired.

[2] In XML vernacular, fields are called either elements or attributes.

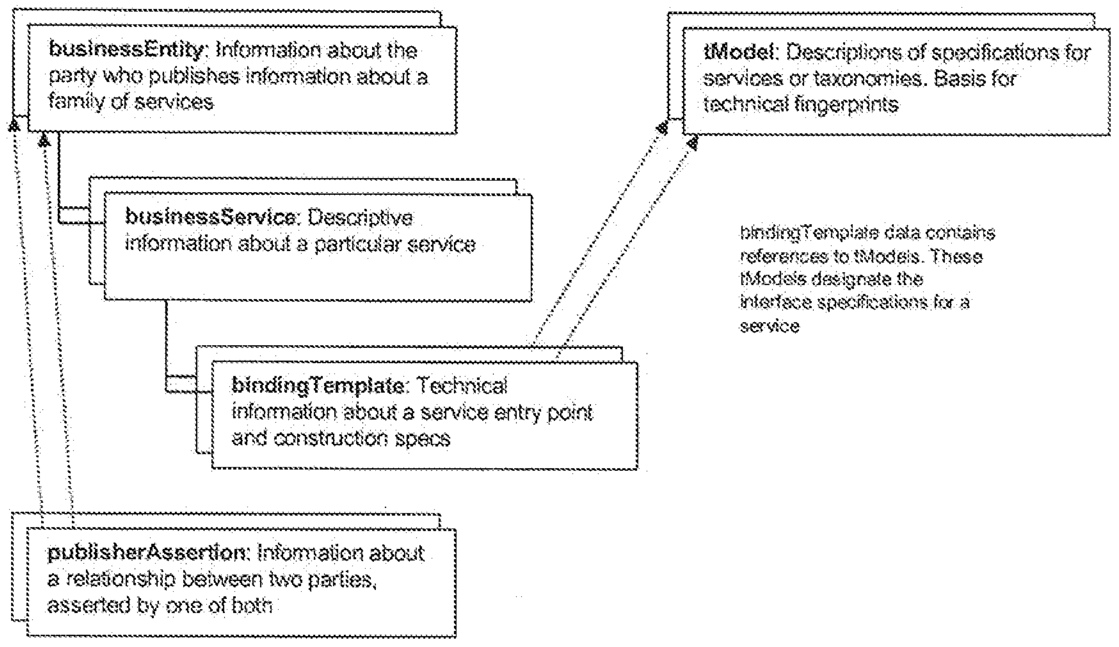

businessEntity: Information about the party who publishes information about a family of services
tModel: Descriptions of specifications for services or taxonomies. Basis for technical fingerprints
businessService: Descriptive information about a particular service
bindingTemplate data contains references to tModels. These tModels designate the interface specifications for a service
bindingTemplate: Technical information about a service entry point and construction specs
publisherAssertion: Information about a relationship between two parties, asserted by one of both

Figure 1

3 Overall Design Principles

Each of the five structure types is used to express specific types of data, arranged in the relationship shown in Figure 1. A particular instance of an individual fact or set of related facts is expressed using XML according to the definition of these core types. For instance, two separate businesses may publish information about the Web services they offer, whether these services are entry points for interfacing with accounting systems, or even services that allow customers to query the status of a factory order. Each business, and the corresponding service descriptions (both logical and technical descriptions) all exist as separate instances of data within a UDDI registry.

3.1 Unique identifiers

The individual facts about a business, its services, technical information, or even information about specifications for services are kept separate, and are accessed individually by way of unique identifiers, or keys. A UDDI registry assigns these unique identifiers when information is first saved, and these identifiers can be used later as keys to access the specific data instances on demand.

Each unique identifier generated by a UDDI registry takes the form of a Universally Unique ID[3] (UUID). Technically, a UUID is a hexadecimal string that has been generated according to a very exacting algorithm that is sufficiently precise as to prevent any two UUIDs from ever being generated in duplicate[4].

[3] The terms "Universally Unique Identifier" (UUID) and "Globally Unique Identifier" (GUID) are used synonymously in technical documentation. In the remainder of this document, the term UUID is used.

[4] The UUID structure and generation algorithm is described in the ISO/IEC 11578:1996 standard (see www.iso.ch).

3.2 Containment

The individual instance data managed by a UDDI registry are sensitive to the parent/child relationships found in the XML schema. This same containment relationship is seen in figure 1 for the core structures. The businessEntity structure contains one or more unique businessService structures. Similarly, individual businessService structures contain specific instances of bindingTemplate data, which in turn contains information that includes pointers to specific instances of tModel structures.

It is important to note that no single instance of a core structure type is ever "contained" by more than one parent structure. This means that only one specific businessEntity structure (identified by its unique key value) will ever contain or be used to express information about a specific instance of a businessService structure (also identified by its own unique key value).

References, on the other hand, operate differently. We can see an example of this in figure 1 where the bindingTemplate structures contain references to unique instances of tModel structures. References can be repeated within any number of the core typed data instances such that many references to a single unique instance are allowed.

Determining what is a reference to an instance of a core data type and what is a key for a core data type within a specific instance is straightforward. There are five core data types, and instances of each of these types are identified by unique keys. The businessKey found within the businessEntity structure is a key, and not a reference. Similarly, the serviceKey and bindingKey values found respectively within the businessService and bindingTemplate structures are keys. The same holds true for the tModelKey value found within the tModel structure. The publisherAssertion's key is logically the concatenation of all of its elements.

References on the other hand, occur in several places, especially for tModels. When tModels are referenced, as seen within a bindingTemplate structure, these occur within a list structure designed for the purpose of holding references to tModels. This list, not being one of the five core data types, is not keyed as an individual instance. Rather, its own identity is derived from the parent structure that contains it – and it cannot be separated. Thus any key values directly contained in structures that are not themselves one of the five core structure types are references. Examples include tModelKey values found in lists within bindingTemplate and categorization and identification schemes – in which context the tModel represents a uniquely identifiable namespace reference and qualifier.

4 Data Structure Notation

Data structures are described by substructure breakdowns in tables of the following form.

Field Name	**Description**	**Data Type**	**Length**
Optional fields are written in normal font	Description of the field's meaning and whether it's • An attribute or an element • Repeatable or not	Possible Data Types include • structure • string • UUID	If the field's data type is string, the field's length is given here in Unicode characters
Required fields are written in **bold** font			

Most of the data structures are also given in their XML Schema representation. Please use the UDDI XML Schema as the definitive technical reference, if needed.

5 The businessEntity structure

The businessEntity structure represents all known information about a business or entity that publishes descriptive information about the entity as well as the services that it offers. From an XML standpoint, the businessEntity is the top-level data structure that accommodates holding descriptive information about a business or entity. Service descriptions and technical information are expressed within a businessEntity by a containment relationship.

5.1 Structure specification

```
<element name="businessEntity" type="uddi:businessEntity" />
<complexType name="businessEntity">
  <sequence>
    <element ref="uddi:discoveryURLs" minOccurs="0" />
    <element ref="uddi:name" maxOccurs="unbounded" />
    <element ref="uddi:description" minOccurs="0" maxOccurs="unbounded" />
    <element ref="uddi:contacts" minOccurs="0" />
    <element ref="uddi:businessServices" minOccurs="0" />
    <element ref="uddi:identifierBag" minOccurs="0" />
    <element ref="uddi:categoryBag" minOccurs="0" />
  </sequence>
  <attribute name="businessKey" type="uddi:businessKey" use="required" />
  <attribute name="operator" type="string" use="optional" />
  <attribute name="authorizedName" type="string" use="optional" />
</complexType>
```

5.2 Substructure breakdown

Field Name	Description	Data Type	Length
businessKey	Attribute. This is the unique identifier for a given instance of a businessEntity structure.	UUID	41
authorizedName	Attribute. This is the recorded name of the individual that published the businessEntity data. This data is generated by the controlling operator and should not be supplied within save_business operations.	string	255
operator	Attribute. This is the certified name of the UDDI registry site operator that manages the master copy of the businessEntity data. The controlling operator records this data at the time data is saved. This data is generated and should not be supplied within save_business operations.	string	255
discoveryURLs	Optional element. This is a list of Uniform Resource Locators (URL) that point to alternate, file based service discovery mechanisms. Each recorded businessEntity structure is automatically assigned a URL that returns the individual businessEntity structure. URL search is provided via find_business call.	structure	
name	Required repeating element. These are the human readable names recorded for the businessEntity, adorned with a unique xml:lang value to signify the language that they are expressed in. Name search is provided via find_business call. Names may not be blank.	string	255

Field Name	Description	Data Type	Length
description	Optional repeating element. One or more short business descriptions. One description is allowed per national language code supplied.	string	255
contacts	Optional element. This is an optional list of contact information.	structure	
businessServices	Optional element. This is a list of one or more logical business service descriptions.	structure	
identifierBag	Optional element. This is an optional list of name-value pairs that can be used to record identifiers for a businessEntity. These can be used during search via find_business.	structure	
categoryBag	Optional element. This is an optional list of name-value pairs that are used to tag a businessEntity with specific taxonomy information (e.g. industry, product or geographic codes). These can be used during search via find_business.	structure	

5.2.1 discoveryURLs

The discoveryURLs structure is used to hold pointers to URL addressable discovery documents. The expected retrieval mechanism for URLs referenced in the data within this structure is HTTP-GET. The expected return document is not defined. Rather, a framework for establishing convention is provided, and two such conventions are defined within UDDI behaviors. It is hoped that other conventions come about and use this structure to accommodate alternate means of discovery.[5]

Field Name	Description	Data Type	Length
discoveryURL	Attribute qualified repeating element holding strings that represent web addressable (via HTTP-GET) discovery documents.	string w/attributes	255

5.2.1.1 discoveryURL

Each individual discovery URL consists of an attribute whose value designates the URL use type convention, and a string, found within the body of the element. Each time a businessEntity structure is saved via a call to save_business, the UDDI Operator Site will generate one URL. The generated URL will point to an instance of either a businessEntity or businessEntityExt structure, and the useType attribute of the discoveryURL will be set to either "businessEntity" or "businessEntityExt" according to the data type found while processing the save_business message. The discoveryURLs collection will be augmented so that it includes this generated URL. This URL can then be used to retrieve a specific instance of a businessEntity, since the XML returned will be formatted as a normal businessDetail message.

[5] An example of an alternate form of service discovery is seen in the ECO Framework as defined by the commerce.net initiative. A convention to provide pointers to ECO discovery entry points could take advantage of the structures provided in discoveryURLs by adopting the useType value "ECO".

Field Name	Description	Data Type	Length
useType	Required attribute that designates the name of the convention that the referenced document follows. Two reserved convention values are "businessEntity" and "businessEntityExt". URLs qualified with these values should point to XML documents of the same type as the useType value.	string	255

Example: An example of the generated data for a given businessEntity might look similar to the following:

```
<discoveryURLs>
   <discoveryURL useType="businessEntity">
   http://www.someOperator?businessKey=BE3D2F08-CEB3-11D3-849F-0050DA1803C0
   </discoveryURL>
<discoveryURLs>
```

5.2.2 name

A businessEntity MAY contain more than one name. Multiple names are useful, for example, in order to specify both the legal name and a known abbreviation of a businessEntity, or in order to support romanization.

When a name is expressed in a specific language (such as the language into which a name has been romanized), it SHOULD carry the xml:lang attribute to signify this. When a name does not have an associated language (such as a neologism not associated with a particular language), the xml:lang attribute SHOULD be omitted.

As is defined in the XML specification, an occurrence of the xml:lang attribute indicates that the content to which it applies (namely the element on which it is found and to all its children, unless subsequently overridden) is to be interpreted as being in a certain natural language. Legal values for such attributes are specified in the IETF standard RFC 1766 and its successors (including, as of the time of the present writing, RFC 3066). As is indicated therein, language values begin with a primary language tag, and are optionally followed by a series of hyphen-delimited sub-tags for country or dialect identification; the tags are not case-sensitive. Examples include: "EN-us", "FR-ca".

The same mechanism applies to the name element within the businessService structure.

5.2.3 contacts

The contacts structure provides a way for information to be registered with a businessEntity record so that someone that finds the information can make human contact for any purpose. Since the information held within the UDDI Operator Sites is freely available, some care should be taken when considering the amount of contact information to register. Electronic mail addresses in particular may be the greatest concern if you are sensitive to receiving unsolicited mail.

The contacts structure itself is a simple collection of contact structures. You'll find that there are many collections in the UDDI Version 2.0 API schema. Like the discoveryURLs structure – which is a container for one or more discoveryURL structures, the contacts structure is a simple container where one or more contact structures reside.

5.2.3.1 contact

The contact structure lets you record contact information for a person. This information can consist of one or more optional elements, along with a person's name. Contact information exists by containment relationship alone, and no mechanisms for tracking individual contact instances is provided by UDDI specifications.

For transliteration purposes (e.g. romanization) the suggested approach is to file multiple contacts.

Field Name	Description	Data Type	Length
useType	Optional attribute that is used to describe the type of contact in freeform text. Suggested examples include "technical questions", "technical contact", "establish account", "sales contact", etc.	string	255
description	Optional element. Zero or more language-qualified[6] descriptions of the reason the contact should be used.	string	255
personName	Required element. Contacts should list the name of the person or name of the job role that will be available behind the contact. Examples of roles include "administrator" or "webmaster".	string	255
phone	Optional repeating element. Used to hold telephone numbers for the contact. This element can be adorned with an optional useType attribute for descriptive purposes.	string w/attributes	50
email	Optional repeating element. Used to hold email addresses for the contact. This element can be adorned with an optional useType attribute for descriptive purposes.	string w/attributes	255
address	Optional repeating element. This structure represents the printable lines suitable for addressing an envelope.	structure	

5.2.3.2 address

The address structure is a simple list of addressLine elements within the address container. Each addressLine element is a simple string. UDDI compliant registries are responsible for preserving the order of any addressLine data provided. Address structures also have three optional attributes. The useType describes the address' type in freeform text. The sortCode values are not significant within a UDDI registry, but may be used by user interfaces that present contact information in some ordered fashion using the values provided in the sortCode attribute. The tModelKey is a tModel reference that specifies that keyName keyValue pairs given by subsequent addressLine elements, if addressLine elements are present at all, are to be interpreted by the address structure associated with the tModel that is referenced. For a description of how to use tModels in order to give the simple addressLine list structure and meaning, see Appendix E: Structured Address Example.

[6] All fields named *description* behave the same way and are subject to the same language identifier rules as described in the XML usage appendix found in the UDDI programmers API specification. Embedded HTML is prohibited in description fields.

Field Name	Description	Data Type	Length
useType	Optional attribute that is used to describe the type of address in freeform text. Suggested examples include "headquarters", "sales office", "billing department", etc.	string	255
sortCode	Optional attribute that can be used to drive the behavior of external display mechanisms that sort addresses. The suggested values for sortCode include numeric ordering values (e.g. 1, 2, 3), alphabetic character ordering values (e.g. a, b, c) or the first n positions of relevant data within the address.	string	10
tModelKey	Optional attribute. This is the unique key reference that implies that the keyName keyValue pairs given by subsequent addressLine elements are to be interpreted by the taxonomy associated with the tModel that is referenced.	string	255 41[7]
addressLine	Optional repeating element containing the actual address in freeform text. If the address element contains a tModelKey, these addressLine elements are to be adorned each with an optional keyName keyValue attribute pair. Together with the tModelKey, keyName and keyValue qualify the addressLine in order to describe its meaning.	string w/attributes	80

5.2.3.3 addressLine

AddressLine elements contain string data with a line length limit of 80 character positions. Each addressLine element can be adorned with two optional descriptive attributes, keyName and keyValue. Both attributes must be present in each address line if a tModelKey is assigned to the address structure. By doing this, the otherwise arbitrary use of address lines becomes structured. Together with the address' tModelKey, the keyName and keyValue qualify the addressLine according to the address structure specified in the overview document of the referenced tModel. See Appendix E for an example how structured addresses can be represented. When no tModelKey is provided for the address structure, the keyName and keyValue attributes can be used without restrictions, for example, to provide descriptive information for each addressLine by using the keyName attribute. Since both the keyName and the keyValue attributes are optional, address line order is significant and will always be returned by the UDDI compliant registry in the order originally provided during a call to save_business.

5.2.4 businessServices

The businessServices structure provides a way for describing information about families of services. This simple collection accessor contains zero or more businessService structures and has no other associated structures.

5.2.5 identifierBag

The identifierBag element allows businessEntity or tModel structures to include information about common forms of identification such as D-U-N-S® numbers, tax identifiers, etc. This data can be used to signify the identity of the businessEntity, or can be used to signify the identity of the publishing party.

[7] The data type for tModelKey allows for using URN values in a later revision. In the current release, the key is a generated UUID. Design work around managing duplicate urn claims will allow user supplied URN keys on tModels in the future.

Including data of this sort is optional, but when used greatly enhances the search behaviors exposed via the *find_xx* messages defined in the UDDI Version 2.0 API Specification. For a full description of the structures involved in establishing an identity, see Appendix A: Using Identifiers.

5.2.6 categoryBag

The categoryBag element allows businessEntity, businessService and tModel structures to be categorized according to any of several available taxonomy based classification schemes. Operator Sites automatically provide validated categorization support for three taxonomies that cover industry codes (via NAICS), product and service classifications (via UNSPC) and geography (via ISO 3166). Including data of this sort is optional, but when used greatly enhances the search behaviors exposed by the *find_xx* messages defined in the UDDI Version 2.0 API Specification. For a full description of structures involved in establishing categorization information, see Appendix B: Using categorization.

6 The businessService structure

The businessService structures each represent a logical service classification. The name of the element includes the term "business" in an attempt to describe the purpose of this level in the service description hierarchy. Each businessService structure is the logical child of a single businessEntity structure. The identity of the containing (parent) businessEntity is determined by examining the embedded businessKey value. If no businessKey value is present, the businessKey must be obtainable by searching for a businessKey value in any parent structure containing the businessService. Each businessService element contains descriptive information in business terms outlining the type of technical services found within each businessService element.

In some cases, businesses would like to share or reuse services, e.g. when a large enterprise publishes separate businessEntity structures. This can be established by using the businessService structure as a *projection* to an already published businessService.

Any businessService projected in this way is not managed as a part of the referencing businessEntity, but centrally as a part of the referenced businessEntity. This means that changes of the businessService by the referenced businessEntity are automatically valid for the service projections done by referencing businessEntity structures.

In order to specify both referenced and referencing businessEntity structures correctly, service projections can only be published by a save_business message with the referencing businessKey present in the businessEntity structure and both the referenced businessKey and the referenced businessService present in the businessService structure.

6.1 Structure Specification

```
<element name="businessService" type="uddi:businessService" />
<complexType name="businessService">
  <sequence>
    <element ref="uddi:name" minOccurs="0" maxOccurs="unbounded" />
    <element ref="uddi:description" minOccurs="0" maxOccurs="unbounded" />
    <element ref="uddi:bindingTemplates" minOccurs="0" />
    <element ref="uddi:categoryBag" minOccurs="0" />
  </sequence>
  <attribute name="serviceKey" type="uddi:serviceKey" use="required" />
  <attribute name="businessKey" type="uddi:businessKey" use="optional" />
</complexType>
```

6.2 Substructure Breakdown

Field Name	Description	Data Type	Length
businessKey	This attribute is optional when the businessService data is contained within a fully expressed parent that already contains a businessKey value. If the businessService data is rendered into XML and has no containing parent that has within its data a businessKey, the value of the businessKey that is the parent of the businessService is required to be provided. This behavior supports the ability to browse through the parent-child relationships given any of the core elements as a starting point. The businessKey may differ from the publishing businessEntity's businessKey to allow service projections.	UUID	41

Field Name	Description	Data Type	Length
serviceKey	This is the unique key for a given businessService. When saving a new businessService structure, pass an empty serviceKey value. This signifies that a UUID value is to be generated. To update an existing businessService structure, pass the UUID value that corresponds to the existing service. If this data is received via an inquiry operation, the serviceKey values may not be blank. When saving a new or updated service projection, pass the serviceKey of the referenced businessService structure.	UUID	41
name	Optional repeating element. These are the human readable names recorded for the businessService, adorned with a unique xml:lang value to signify the language that they are expressed in. Name search is provided via find_service call. Names may not be blank. When saving a new or updated service projection, pass the exact name of the referenced businessService, here.	string	255
description	Optional element. Zero or more language-qualified text descriptions of the logical service family.	string	255
bindingTemplates	This structure holds the technical service description information related to a given business service family.	structure	
categoryBag	Optional element. This is an optional list of name-value pairs that are used to tag a businessService with specific taxonomy information (e.g. industry, product or geographic codes). These can be used during search via find_service. See categoryBag under businessEntity for a full description.	structure	

6.2.1 bindingTemplates

The bindingTemplates structure is a container for zero or more bindingTemplate structures. This simple collection accessor has no other associated structure.

7 The bindingTemplate structure

Technical descriptions of Web services are accommodated via individual contained instances of bindingTemplate structures. These structures provide support for determining a technical entry point or optionally support remotely hosted services, as well as a lightweight facility for describing unique technical characteristics of a given implementation. Support for technology and application specific parameters and settings files are also supported.

Since UDDI's main purpose is to enable description and discovery of Web Service information, it is the bindingTemplate that provides the most interesting technical data.

Each bindingTemplate structure has a single logical businessService parent, which in turn has a single logical businessEntity parent.

7.1 Structure specification

```
<element name="bindingTemplate" type="uddi:bindingTemplate" />
<complexType name="bindingTemplate">
  <sequence>
    <element ref="uddi:description" minOccurs="0" maxOccurs="unbounded" />
    <choice>
      <element ref="uddi:accessPoint" />
      <element ref="uddi:hostingRedirector" />
    </choice>
    <element ref="uddi:tModelInstanceDetails" />
  </sequence>
  <attribute name="serviceKey" type="uddi:serviceKey" use="optional" />
  <attribute name="bindingKey" type="uddi:bindingKey" use="required" />
</complexType>
```

7.2 Substructure breakdown

Field Name	Description	Data Type	Length
bindingKey	This is the unique key for a given bindingTemplate. When saving a new bindingTemplate structure, pass an empty bindingKey value. This signifies that a UUID value is to be generated. To update an existing bindingTemplate structure, pass the UUID value that corresponds to the existing bindingTemplate instance. If this data is received via an inquiry operation, the bindingKey values may not be blank.	UUID	41
serviceKey	This attribute is optional when the bindingTemplate data is contained within a fully expressed parent that already contains a serviceKey value. If the bindingTemplate data is rendered into XML and has no containing parent that has within its data a serviceKey, the value of the serviceKey that is the ultimate containing parent of the bindingTemplate is required to be provided. This behavior supports the ability to browse through the parent-child relationships given any of the core elements as a starting point.	UUID	41

Field Name	Description	Data Type	Length
description	Optional repeating element. Zero or more language-qualified text descriptions of the technical service entry point.	string	255
accessPoint	Required attribute qualified element[8]. This element is a text field that is used to convey the entry point address suitable for calling a particular Web service. This may be a URL, an electronic mail address, or even a telephone number. No assumptions about the type of data in this field can be made without first understanding the technical requirements associated with the Web service[9].	string w/attributes	255
hostingRedirector	Required element if *accessPoint* not provided. This element is adorned with a bindingKey attribute, giving the redirected reference to a different bindingTemplate. If you query a bindingTemplate and find a hostingRedirector value, you should retrieve that bindingTemplate and use it in place of the one containing the hostingRedirector data.	empty w/attributes	
tModelInstanceDetails	This structure is a list of zero or more tModelInstanceInfo elements. This data, taken in total, should form a distinct fingerprint that can be used to identify compatible services.	structure	

7.2.1 accessPoint

The accessPoint element is an attribute-qualified pointer to a service entry point. The notion of service at the metadata level seen here is fairly abstract and many types of entry points are accommodated.

A single attribute is provided (named URLType). The purpose of the URLType attribute is to facilitate searching for entry points associated with a particular type of entry point. An example might be a purchase order service that provides three entry points, one for HTTP, one for SMTP, and one for FAX ordering. In this example, we'd find a businessService element that contains three bindingTemplate entries, each with identical data with the exception of the accessPoint value and URLType value.

The valid values for URLType are:

- **mailto**: designates that the accessPoint string is formatted as an electronic mail address reference, for example, mailto:purch@fabrikam.com.
- **http**: designates that the accessPoint string is formatted as an HTTP compatible Uniform Resource Locator (URL), for example, http://www.fabrikam.com/purchasing.
- **https**: designates that the accessPoint string is formatted as a secure HTTP compatible URL, for example https://www.fabrikam.com/purchasing.
- **ftp**: designates that the accessPoint string is formatted as a FTP directory address, for example ftp://ftp.fabrikam.com/public.

[8] One of accessPoint or hostingRedirector is required.

[9] The content of the structure named tModelInstanceDetails that is found within a bindingTemplate structure serves as a technical fingerprint. This fingerprint is a series of references to uniquely keyed specifications and/or concepts. To build a new service that is compatible with a tModel, the specifications must be understood. To register a service compatible with a specification, reference a tModelKey within the tModelInstanceDetails data for a bindingTemplate instance.

- **fax**: designates that the accessPoint string is formatted as a telephone number that will connect to a facsimile machine, for example 1 425 555 5555.
- **phone**: designates that the accessPoint string is formatted as a telephone number that will connect to human or suitable voice or tone response based system, for example 1 425 555 5555.
- **other**: designates that the accessPoint string is formatted as some other address format. When this value is used, one or more of the tModel signatures found in the tModelInstanceInfo collection must imply that a particular format or transport type is required.

7.2.2 hostingRedirector

The hostingRedirector element is used to designate that a bindingTemplate entry is a pointer to a different bindingTemplate entry. The value in providing this facility is seen when a business or entity wants to expose a service description (e.g. advertise that they have a service available that suits a specific purpose) that is actually a service that is described in a separate bindingTemplate record. This might occur when a service is remotely hosted (hence the name of this element), or when many service descriptions could benefit from a single service description.

The hostingRedirector element has a single attribute and no element content. The attribute is a bindingKey value that is suitable within the same UDDI registry instance for querying and obtaining the bindingDetail data that is to be used.

More on the hostingRedirector can be found in the appendices for the UDDI Version 2.0 API Specification.

7.2.3 tModelInstanceDetails

This structure is a simple accessor container for one or more tModelInstanceInfo structures. When taken as a group, the data that is presented in a tModelInstanceDetails structure forms a technically descriptive fingerprint by virtue of the unordered list of tModelKey references contained within this structure. What this means in English is that when someone registers a bindingTemplate (within a businessEntity structure), it will contain one or more references to specific and identifiable specifications that are implied by the tModelKey values provided with the registration. During an inquiry for a service, an interested party could use this information to look for a specific bindingTemplate that contains a specific tModel reference, or even a set of tModel references. By registering a specific fingerprint in this manner, a software developer can readily signify that they are compatible with the specifications implied in the tModelKey elements exposed in this manner.

7.2.3.1 tModelInstanceInfo

A tModelInstanceInfo structure represents the bindingTemplate instance specific details for a single tModel by reference.

Field Name	Description	Data Type	Length
tModelKey	Required Attribute. This is the unique key reference that implies that the service being described has implementation details that are specified by the specifications associated with the tModel that is referenced	string	255
description	Optional repeating element. This is one or more language qualified text descriptions that designate what role a tModel reference plays in the overall service description.	string	255

Field Name	Description	Data Type	Length
instanceDetails	Optional element. This element can be used when tModel reference specific settings or other descriptive information are required to either describe a tModel specific component of a service description or support services that require additional technical data support (e.g. via settings or other handshake operations)	structure	

7.2.3.2 instanceDetails

This structure holds service instance specific information that is required to either understand the service implementation details relative to a specific tModelKey reference, or to provide further parameter and settings support. If present, this element should not be empty. Because no single contained element is required in the schema description, this rule is called out here for clarity.

Field Name	Description	Data Type	Length
description	Optional repeating element. This language-qualified text element is intended for holding a description of the purpose and/or use of the particular instanceDetails entry.	string	255
overviewDoc	Optional element. Used to house references to remote descriptive information or instructions related to proper use of a bindingTemplate technical sub-element.	structure	
instanceParms	Optional element. Used to contain settings parameters or a URL reference to a file that contains settings or parameters required to use a specific facet of a bindingTemplate description. If used to house a URL pointer to a file, the suggested format is URL that is suitable for retrieving the settings or parameters via HTTP-GET.	string	255

7.2.3.3 overviewDoc

This optional structure is provided as a placeholder for metadata that describes overview information about a particular tModel use within a bindingTemplate.

Field Name	Description	Data Type	Length
description	Optional repeating element. This language-qualified string is intended to hold a short descriptive overview of how a particular tModel is to be used.	string	255
overviewURL	Optional element. This string data element is to be used to hold a URL reference to a long form of an overview document that covers the way a particular tModel specific reference is used as a component of an overall web service description. The suggested format is a URL that is suitable for retrieving an HTML based description via a web browser or HTTP-GET operation.	string	255

8 The tModel structure

Being able to describe a Web service and then make the description meaningful enough to be useful during searches is an important UDDI goal. Another goal is to provide a facility to make these descriptions useful enough to learn about how to interact with a service that you don't know much about. In order to do this, there needs to be a way to mark a description with information that designates how it behaves, what conventions it follows, or what specifications or standards the service is compliant with. Providing the ability to describe compliance with a specification, concept, or even a shared design is one of the roles that the tModel structure fills.

The tModel structure takes the form of keyed metadata (data about data). In a general sense, the purpose of a tModel within the UDDI registry is to provide a reference system based on abstraction. Thus, the kind of data that a tModel represents is pretty nebulous. In other words, a tModel registration can define just about anything, but in the current revision, two conventions have been applied for using tModels: as sources for determining compatibility and as keyed namespace references.

The information that makes up a tModel is quite simple. There's a key, a name, an optional description, and then a URL that points somewhere – presumably somewhere where the curious can go to find out more about the actual concept represented by the metadata in the tModel itself.

8.1 Two main uses

There are two places within a businessEntity registration that you'll find references to tModels. In this regard, tModels are special. Whereas the other data within the businessEntity (e.g. businessService and bindingTemplate data) exists uniquely with one uniquely keyed instance as a member of one unique parent businessEntity, tModels are used as references. This means that you'll find references to specific tModel instances in many businessEntity structures.

8.1.1 Defining the technical fingerprint

The primary role that a tModel plays is to represent a technical specification. An example might be a specification that outlines wire protocols, interchange formats and interchange sequencing rules. Examples can be seen in the RosettaNet Partner Interface Processes [10] specification, the Open Applications Group Integration Specification[11] and various Electronic Document Interchange (EDI) efforts.

Software that communicates with other software across some communication medium invariably adheres to some pre-agreed specifications. In situations where this is true, the designers of the specifications can establish a unique technical identity within a UDDI registry by registering information about the specification in a tModel.

Once registered in this way, other parties can express the availability of Web services that are compliant with a specification by simply including a reference to the tModel identifier (called a tModelKey) in their technical service descriptions bindingTemplate data.

This approach facilitates searching for registered Web services that are compatible with a particular specification. Once you know the proper tModelKey value, you can find out whether a particular business or entity has registered a Web service that references that tModel key. In this way, the tModelKey becomes a technical fingerprint that is unique to a given specification.

[10] See www.rosettanet.org

[11] See www.openapplications.org

8.1.2 Defining an abstract namespace reference

The other place where tModel references are used is within the identifierBag, categoryBag, address and publisherAssertion structures that are used to define organizational identity and various classifications. Used in this context, the tModel reference represents a relationship between the keyed name-value pairs to the super-name, or namespace within which the name-value pairs are meaningful.

An example of this can be seen in the way a business or entity can express the fact that their US tax code identifier (which they are sure they are known by to their partners and customers) is a particular value. To do this, let's assume that we find a tModel that is named "US Tax Codes", with a description "United States business tax code numbers as defined by the United States Internal Revenue Service". In this regard, the tModel still represents a specific concept – but instead of being a technical specification, it represents a unique area within which tax code ID's have a particular meaning.

Once this meaning is established, a business can use the tModelKey for the tax code tModel as a unique reference that qualifies the remainder of the data that makes up an entry in the identifierBag data.

To get things started, the UDDI Operator Sites have registered a number of useful tModels, including NAICS (an industry code taxonomy), UNSPC (a product and service category code taxonomy), and ISO 3166 (a geographical region code taxonomy).

8.2 Structure specification

```
<element name="tModel" type="uddi:tModel" />
<complexType name="tModel">
  <sequence>
    <element ref="uddi:name" />
    <element ref="uddi:description" minOccurs="0" maxOccurs="unbounded" />
    <element ref="uddi:overviewDoc" minOccurs="0" />
    <element ref="uddi:identifierBag" minOccurs="0" />
    <element ref="uddi:categoryBag" minOccurs="0" />
  </sequence>
  <attribute name="tModelKey" type="uddi:tModelKey" use="required" />
  <attribute name="operator" type="string" use="optional" />
  <attribute name="authorizedName" type="string" use="optional" />
</complexType>
```

8.3 Substructure breakdown

Field Name	Description	Data Type	Length
tModelKey	Required Attribute. This is the unique key for a given tModel structure. When saving a new tModel structure, pass an empty tModelKey value. This signifies that a UUID value is to be generated. To update an existing tModel structure, pass the tModelKey value that corresponds to an existing tModel instance.	string	255
authorizedName	Attribute. This is the recorded name of the individual that published the tModel data. This data is calculated by the controlling operator and should not be supplied within save_tModel operations.	string	255
operator	Attribute. This is the certified name of the UDDI registry site operator that manages the master copy of the tModel data. The controlling operator records this data at the time data is saved. This data is calculated and should not be supplied within save_tModel operations.	string	255

Field Name	Description	Data Type	Length
name	Required element. This is the name recorded for the tModel. Name search is provided via find_tModel call. Names may not be blank, and should be meaningful to someone who looks at the tModel. The name should be formatted as a URI and, as a consequence, the xml:lang attribute of the name element should not be used.	string	255
description	Optional repeating element. One or more short language-qualified descriptions. One description is allowed per national language code supplied.	string	255
overviewDoc	Optional element. Used to house references to remote descriptive information or instructions related to the tModel. See the substructure breakdown for overviewDoc in section The bindingTemplate structure.	structure	
identifierBag	Optional element. This is an optional list of name-value pairs that can be used to record identification numbers for a tModel. These can be used during search via find_tModel. See the full description of this element in the businessEntity section of this document and in Appendix A: Using Identifiers.	structure	
categoryBag	Optional element. This is an optional list of name-value pairs that are used to tag a tModel with specific taxonomy information (e.g. industry, product or geographic codes). These can be used during search via find_tModel. See the full description of this element in the businessEntity section of this document and in Appendix B: Using categorization	structure	

9 The publisherAssertion structure

Many businesses, like large enterprises or marketplaces, are not effectively represented by a single businessEntity, since their description and discovery are likely to be diverse. As a consequence, several businessEntity structures can be published, representing individual subsidiaries of a large enterprise or individual participants of a marketplace. Nevertheless, they still represent a more or less coupled community and would like to make some of their relationships visible in their UDDI registrations. Therefore, two related businesses use the *xx*_publisherAssertion messages, publishing assertions of business relationships.

In order to eliminate the possibility that one publisher claims a relationship between both businesses that is in fact not reciprocally recognized, both publishers have to agree that the relationship is valid by publishing their own publisherAssertion. Therefore, both publishers have to publish exactly the same information. When this happens, the relationship becomes visible. More detailed information is given in the appendices for the UDDI Version 2.0 API Specification.

In the case that a publisher is responsible for both businesses, the relationship automatically becomes visible after publishing just one of both assertions that make up the relationship.

The publisherAssertion structure consists of the three elements fromKey (the first businessKey), toKey (the second businessKey) and keyedReference. The keyedReference designates the asserted relationship type in terms of a keyName keyValue pair within a tModel, uniquely referenced by a tModelKey. All three parts of the keyedReference – the tModelKey, the keyName, and the keyValue – are mandatory in this context. Empty (zero length) keyName and keyValue elements are permitted.

9.1 Structure Specification

```
<element name="publisherAssertion" type="uddi:publisherAssertion" />
<complexType name="publisherAssertion">
  <sequence>
    <element ref="uddi:fromKey" />
    <element ref="uddi:toKey" />
    <element ref="uddi:keyedReference" />
  </sequence>
</complexType>
```

9.2 Substructure Breakdown

Field Name	Description	Data Type	Length
fromKey	Required element. This is the unique key reference to the first businessEntity the assertion is made for.	UUID	41
toKey	Required element. This is the unique key reference to the second businessEntity the assertion is made for.	UUID	41
keyedReference	Required element. This designates the relationship type the assertion is made for, represented by the included tModelKey and described by the included keyName keyValue pair.	empty w/attributes	

10 Appendix A: Using Identifiers

10.1 The identifier dilemma

One of the design goals associated with the UDDI registration data is the ability to mark information with identifiers. The purpose of identifiers in the UDDI registration data is to allow others to find the published information using more formal identifiers such as D-U-N-S® numbers[12], Global Location Numbers (GLN)[13], tax identifiers, or any other kind of organizational identifiers, regardless of whether these are private or shared.

When you look at an identifier, such as a D-U-N-S® number, it is not always immediately apparent what the identifier represents. For instance, consider the following identifier:

```
123-45-6789
```

Standing alone, we could try and guess what this combination of digits and formatting characters implies. However, if we knew that this was a United States Social Security number, we would then have a better context and understand that this string, while still not clear, at least identifies one or more persons, perhaps even a living one. Expressed as a name / value pair, the identifier might then look like the following:

```
United States Social Security Number, 123-45-6789
```

Even with this new information, a search mechanism based on loosely qualified pairs (name of identifier type, identifier value), two different parties might spell or format either part of the information differently, and with the end result being a diminished value for searching.

The goal, of course, is to define a simple mechanism that disambiguates the conceptual meanings behind identifiers and exposes them in ways that are reliable and predictable enough to use, and yet are simple enough structurally to be easy to understand and extend.

10.2 Identifier characteristics

When we look at various types of simple identifiers, some common desirable characteristics become evident. In general terms, a system of identifiers that are used to facilitate searching need to be:

- **Resolvable**: Identifiers can be used in a way that allows the meaning of the identifier to be determined. For instance, a popular business identifier mechanism is provided by Dun & Bradstreet in the form of D-U-N-S® numbers. When you know an organization's D-U-N-S® number, you can use this to reliably distinguish one organization from another.
- **Distinguishable**: Identifiers can be used in a way that you can tell what kind of identifier is being used, or you can specify what kind of identifier you are using to search for something. This means you can tell that two identifiers are the same kind of identifier or are different types (e.g. two D-U-N-S® numbers, versus a tax identifier or an organizational membership number.)
- **Extensible**: The way that searchable identifiers are defined should be easy to extend so that anyone can register another type of identifier without having to create costly or difficult infrastructure. The search mechanisms that use identifiers should be able to accommodate newly registered types without any changes to software, and anyone should be able to start using the new types immediately.

With this in mind, let's look at the way that identifiers are used in the UDDI data structures.

[12] D-U-N-S® Numbers are provided by Dun & Bradstreet. See http://www.dnb.com.

[13] The Global Location Number system is defined in the EAN UCC system (http://www.ean-int.org/locations.html).

10.2.1 Using identifiers

Instead of defining a simple property where you could attach a keyword or a simple identifier field, UDDI defines the notion of annotating or attaching identifiers to data. Two of the core data types specified by UDDI provide a structure to support attaching identifiers to data. These are the businessEntity and the tModel structures. By providing a placeholder for attaching identifiers to these two root data types, any number of identifiers can be used for a variety of purposes.

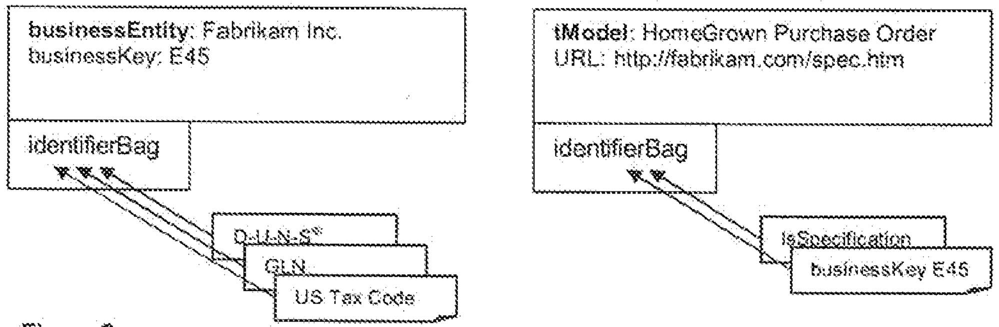

businessEntity: Fabrikam Inc.
businessKey: E45
identifierBag
D-U-N-S®
GLN
US Tax Code
tModel: HomeGrown Purchase Order
URL: http://fabrikam.com/spec.htm
identifierBag
IsSpecification
businessKey E45

Figure 2

In figure 2 we see that businessEntity and tModel structures both have a placeholder element named identifierBag[14]. This structure is a general-purpose placeholder for any number of distinct identifiers. In this example, we see five types of identifiers in use in a way that accommodates the kinds of searching that might be required to locate businesses or tModels.

For instance, it is likely that someone who wants to find the types of technical Web services that are exposed by a given business would search by a business identifier. Used in this way, identifiers can represent business identifier types. In the example shown in figure 2, we see that the individual who registered the businessEntity data specified a D-U-N-S® number, a Global Location Number, and a US Tax Code identifier[15].

On the other hand, since a tModel is a fairly abstract concept, I might care more that a tModel represents an identifier, and that it was registered by a particular businessEntity. In the example in figure 2, we have shown some more abstract identifier types and can tell that the tModel that describes the way that Fabrikam's purchasing Web service has been marked with information that identifies the data as being related to the businessEntity record with the theoretical businessKey value E45. A second identifier marks the tModel as a specification.

Two identifier types have been identified and made a core part of the UDDI Operator registries, so far. These are the Dun & Bradstreet D-U-N-S® numbers and the Thomas Register supplier IDs[16].

Identifier Name	**tModel name**
D-U-N-S	dnb-com:D-U-N-S
Thomas Register	thomasregister-com:supplierID

10.2.2 Structure Specification

```
<element name="identifierBag" type="uddi:identifierBag" />
```

[14] The term "bag" is from the object design naming convention that places collections of like things within an outer container. From outside, it behaves like a bag – that is has a collection of things in it. To see what's in it, you have to look inside.

[15] In the diagram, the actual name/value properties were abbreviated for the sake of simplicity.

[16] See http://www.thomasregister.com.

```
<complexType name="identifierBag">
  <sequence>
    <element ref="uddi:keyedReference" maxOccurs="unbounded" />
  </sequence>
</complexType>
```

From this structure definition we see that an identifier bag is an element that holds zero or more instances of something called a keyedReference. When we look at that structure, we see:

```
<element name="keyedReference" type="uddi:keyedReference" />
<complexType name="keyedReference">
  <attribute name="tModelKey" type="uddi:tModelKey" use="optional" />
  <attribute name="keyName" type="string" use="optional" />
  <attribute name="keyValue" type="string" use="required" />
</complexType>
```

Upon examining this, we see a general-purpose structure for a name-value pair, with one curious additional reference to a tModel structure. It is this extra attribute that makes the identifier scheme extensible by allowing tModels to be used as conceptual namespace qualifiers.

Understanding this, it then should be easy to see how the example in figure 2 functions. Assuming that the identifiers were fully defined, the five types shown would each reference one of five different tModels. Using the information we've learned already from the discussion of the tModel structure in this document and related texts, we should then be able to see how the tModel structure is useful as a general purpose concept registry with specific UDDI emphasis on the concepts of software specifications, identification schemes, and as we see in The publisherAssertion structure and the next appendix, as a way to define a general taxonomy namespace key.

The net result is that you can register a tModel to represent an idea, and then use a reference to that tModel as part of a general discovery mechanism that allows unknown facts to be discovered and explained.

11 Appendix B: Using categorization

Categorization and the ability to voluntarily assign category information to data in the UDDI distributed registry was a key design goal. Without categorization and the ability to specify that information be tangentially related to some well-known industry, product or geographic categorization code set, locating data within the UDDI registry would prove to be too difficult.

At the same time, it is impractical to assume that the UDDI registry will be useful for general-purpose business search. With a projected near-term population of several hundred thousand to million distinct entities, it is unlikely that searching for businesses that satisfy a particular set of criteria will yield a manageably sized result set. For example, suppose we searched for all businesses that have classified themselves with a particular industry code – retail. Even if we searched within this specific industry classification, the breadth of the category makes it likely that we'll find tens of thousands of companies that are retailers or in some way think of themselves as belonging to a retail category.

Secondary considerations include the accuracy with which categories are applied and the exact value match nature of the UDDI categorization facility. When you register a specific category along with your UDDI registration data, only people searching for that exact category will find your results. For example, in the case where one business marks itself as "retail – pet-food", and another simply uses "retail", the specialization and generalization across categories of any particular categorization scheme or taxonomy is not known to the UDDI search facility.

More intelligent search facilities are required that have some a priori knowledge of the meanings of specific categories and that provide the ability to cross-reference across related categories. Such is the role of more traditional search engines. The design of UDDI allows simplified forms of searching and allows the parties that publish data about themselves, and their advertised Web services to voluntarily provide categorization data that can be used by richer search facilities that will be created above the UDDI technical layer.

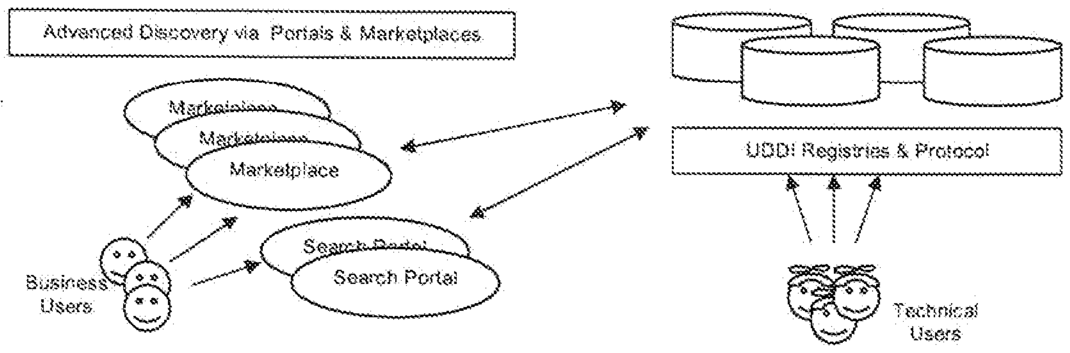


Figure 3

In figure 3 we see the tiered search concept illustrated. The role of search portals and marketplaces will support the business level search facilities for such activities as finding partners with products in a certain price range or availability, or finding high quality partners with good reputations. The data in UDDI is not sufficient to accommodate this because of the cross category issues associated with high volumes and voluntary classification.

11.1 Structure Specification

```
<element name="categoryBag" type="uddi:categoryBag" />
<complexType name="categoryBag">
  <sequence>
    <element ref="uddi:keyedReference" maxOccurs="unbounded" />
  </sequence>
</complexType>
```

From this structure definition we see that categoryBag is an element that holds zero or more instances of keyedReference elements. This was described in the section on identifiers (Appendix A: Using Identifiers) and the basic structure is used in the same way.

Three categorization taxonomies have been identified and made a core part of the UDDI Operator registries, so far. These are the North American Industry Classification System (NAICS)[17], Universal Standard Products and Services Classification (UNSPSC)[18], and ISO 3166[19], the international standard for geographical regions, including codes for countries and first-level administrative subdivisions of countries. A fourth category is also defined – named "Other Taxonomy" – for general-purpose keyword type classification[20].

The tModel names for these taxonomies are

Taxonomy Name	**tModel name**
NAICS	ntis-gov:naics:1997
UNSPSC 3.1	unspsc-org:unspsc:3-1 (deprecated)
UNSPSC	unspsc-org:unspsc
ISO 3166	uddi-org:iso-ch:3166:1999
Other Taxonomy	uddi-org:general_keywords

[17] See http://www.census.gov/epcd/www/naics.html.

[18] See http://www.unspsc.org

[19] See http://www.din.de/gremien/nas/nabd/iso3166ma.

[20] Operator Sites are allowed to promote invalid category entries, or entries that are otherwise rejected by the category classification services, to this miscellaneous taxonomy.

12 Appendix C: Response message reference

All of the messages defined in the UDDI Version 2.0 API Specification return response messages upon successful completion. These structures are defined here for reference purposes. All of the structures shown will appear within SOAP 1.1 compliant envelope structures according to the specifications described in the appendices for the UDDI Version 2.0 API Specification. Only the SOAP <body> element contents are shown in the examples in this section.

12.1 assertionStatusReport

This message returns zero or more assertionStatusItem structures in response to a get_assertionStatusReport inquiry message.

12.1.1 Sample

```
<assertionStatusReport generic="2.0" operator="uddi.someoperator" xmlns="urn:uddi-org:api_v2">
        <assertionStatusItem completionStatus="status:toKey_incomplete">
                <fromKey>F5E65…</fromKey>
                <toKey>A237B…</toKey>
                <keyedReference tModelKey="uuid:F5E65…" keyName="Subsidiary" keyValue="1"
                </keyedReference>
                <keysOwned>
                        <fromKey>F5E65</fromKey>
                </keysOwned>
        </assertionStatusItem>
        [<assertionStatusItem/>…]
</assertionStatusReport>
```

This message reports all complete and incomplete assertions and serves an administrative use including the determination if there are any outstanding, incomplete assertions about relationships involving businesses the publisher account is associated with.

Since the publisher who was authenticated by the get_assertionStatusReport message can manage several businesses, the assertionStatusReport message shows the assertions made for all businesses managed by the publisher.

While the elements fromKey, toKey and keyedReference together identify the assertion whose status is being reported on, the keysOwned element designates those businessKeys the publisher manages.

An assertion is complete only if the completionStatus attribute says so, that is, having a value "status:complete". If completionStatus has a value "status:toKey_incomplete" or "status:fromKey_incomplete", the party who controls the businessEntity referenced by the toKey or the fromKey has not made a matching assertion, yet. In the example we can see that the party who controls the businessEntity with the businessKey A237B… has not made a matching assertion to the one found in the assertionStatusItem, made by the party who controls the businessEntity with the businessKey F5E65… .

12.2 authToken

This message returns the authentication information that should be used in subsequent calls to the publishers API messages.

12.2.1 Sample

```
<authToken generic="2.0" operator="uddi.someoperator" xmlns="urn:uddi-org:api_v2" >
        <authInfo>some opaque token value</authInfo>
</authToken>
```

The authToken message contains a single authInfo element that contains an access token that is to be passed back in all Publisher's API messages that change data. This message is always returned using SSL encryption as a synchronous response to the get_authToken message.

12.3 bindingDetail

This message returns specific bindingTemplate information in response to a get_bindingDetail or find_binding inquiry message.

12.3.1 Sample

```
<bindingDetail generic="2.0" operator="uddi.someoperator" truncated="true"
               xmlns="urn:uddi-org:api_v2">
      <bindingTemplate bindingKey="F5E65…" serviceKey="E4D6…" >
               …
      </bindingTemplate>
      [<bindingTemplate/>…]
</bindingDetail>
```

In this message, one or more bindingTemplate structures are returned according to the data requested in the request message. The serviceKey attributes are always returned when bindingTemplate data is packaged in this way. The truncated flag shown in the example indicates that not all of the requested data was returned due to an unspecified processing limit. Ordinarily, the truncated flag is not included unless the result set has been truncated.

12.4 businessDetail

This message returns one or more complete businessEntity structures in response to a get_businessDetail inquiry message.

12.4.1 Sample

```
<businessDetail generic="2.0" operator="uddi.sourceOperator" truncated="true"
               xmlns="urn:uddi-org:api_v2">
      <businessEntity businessKey="F5E65…" authorizedName="J. Doe"
               operator="uddi.publishingOperator" >
               …
      </businessEntity>
      [<businessEntity/>…]
</businessDetail>
```

In this message, we see that the businessEntity contains the proper output information (e.g. authorizedName, and operator). The two operator attributes shown in the businessDetail element and the businessEntity element reflect the distinguished name of the Operator Site providing the response message and the distinguished name of the operator where the data is controlled, respectively. Additionally, notice the name of the person who registered the data shown in the authorizedName attribute.

12.5 businessDetailExt

This message returns one or more complete businessEntityExt structures in response to a get_businessDetailExt inquiry message. This is the same data returned by the businessDetail messages, but is provided for consistency with third party extensions to businessEntity information.

12.5.1 Sample

```
<businessDetailExt generic="2.0" operator="uddi.sourceOperator" truncated="true"
               xmlns="urn:uddi-org:api_v2">
      <businessEntityExt>
               <businessEntity businessKey="F5E65…" authorizedName="J. Doe"
                         operator="uddi.publishingOperator" >
```

```
                …
                </businessEntity>
        <businessEntityExt>
        [<businessEntityExt/>…]
</businessDetail>
```

The message API design allows third party registries (e.g. non-operator sites) to implement the UDDI Version 2.0 API Specifications while at the same time extending the details collected in a way that will not break tools that are written to UDDI specifications. Operator Sites are required to support the *Ext* form of the businessDetail message for compatibility with tools, but are not allowed to manage extended data.

12.6 businessList

This message returns zero or more businessInfo structures in response to a find_business inquiry message. BusinessInfo structures are abbreviated versions of businessEntity data suitable for populating lists of search results in anticipation of further "drill-down" detail inquiries.

12.6.1 Sample

```
<businessList generic="2.0" operator="uddi.sourceOperator" truncated="true"
                xmlns="urn:uddi-org:api_v2">
        <businessInfos>
                <businessInfo businessKey="F5E65…" >
                        <name>My Company</name>
                        <serviceInfos>
                                <serviceInfo serviceKey="3D45…">
                                        <name>Purchase Orders</name>
                                </serviceInfo>
                        </serviceInfos>
                </businessInfo>
                [<businessInfo/>…]
        </businessInfos>
</businessList>
```

This message returns overview data in the form of zero or more businessInfo structures. Each businessInfo structure contains company name and optional description data, along with a collection element named serviceInfos that in turn can contain one or more serviceInfo structures[21]. Notice that the businessKey attribute is not expressed in the serviceInfo structure due to the fact that this information is available from the containing businessInfo structure.

Please note that since a serviceInfo structure can represent a projection to a deleted businessService, the name element within the serviceInfo structure is optional (see section 4.4.13.3 of the V2 API Specification on deleting projected services).

12.7 publisherAssertions

This message returns zero or more publisherAssertion structures in response to a set_publisherAssertions or a get_publisherAssertions publishing message.

12.7.1 Sample

```
<publisherAssertions generic="2.0" operator="uddi.someoperator" authorizedName="J. Doe"
                xmlns="urn:uddi-org:api_v2">
        <publisherAssertion>
                <fromKey>F5E65…</fromKey>
                <toKey>A237B…</toKey>
                <keyedReference tModelKey="uuid:3AD5…" keyName="Holding Company"
                        keyValue="parent-child"
                </KeyedReference>
```

[21] Refer to the UDDI XML schema for structure details.

```
        </publisherAssertion>
        [<publisherAssertion/>…]
</publisherAssertions>
```

This message returns all assertions made by the publisher who was authenticated in the preceding set_publisherAssertions or the get_publisherAssertions message.

12.8 registeredInfo

This message returns overview information that is suitable for identifying all businessEntity and tModel data published by the requester. Provided as part of the Publisher's API message set, this information is only provided when requested via a get_registeredInfo message over an SSL connection.

12.8.1 Sample

```
<registeredInfo generic="2.0" operator="uddi.sourceOperator" [truncated="false"]
                xmlns="urn:uddi-org:api_v2">
        <businessInfos>
                <businessInfo businessKey="F5E65…" >
                        <name>My Company</name>
                        <serviceInfos>
                                <serviceInfo serviceKey="3D45…">
                                        <name>Purchase Orders</name>
                                </serviceInfo>
                        </serviceInfos>
                </businessInfo>
                [<businessInfo/>…]
        <businessInfos>
        <tModelInfos>
                <tModelInfo tModelKey="uuid:34D5…">
                        <name>Proprietary XML purchase order</name>
                </tModelInfo>
                [<tModelInfo/>…]
        </tModelInfos>
</registeredInfo>
```

This message contains overview data about business and tModel information published by a given publisher. This information is sufficient for driving tools that display lists of registered information and then provide drill-down features. This is the recommended structure for use after a network problem results in an unknown status of saved information.

12.9 relatedBusinessesList

This message returns zero or more relatedBusinessInfo structures in response to a find_relatedBusinesses inquiry message.

12.9.1 Sample

```
<relatedBusinessesList generic="2.0" operator="uddi.someoperator" [truncated="false"]
                xmlns="urn:uddi-org:api_v2">
        <businessKey>F5E65…</businessKey>
        <relatedBusinessInfos>
                <relatedBusinessInfo>
                        <businessKey>A237B…</businessKey>
                        <name>Matt's Garage</name>
                        <description>Car services in …</description>
                        <sharedRelationships direction="toKey">
                                <keyedReference tModelKey="uuid:807A2…"
                                        keyName="Subsidiary"
                                        keyValue="parent-child">
                                [<keyedReference/>…]
                        </sharedRelationships>
                </relatedBusinessInfo>
                [<relatedBusinessInfo/>…]
        </relatedBusinessInfos>
```

```
</relatedBusinessesList>
```

For the businessEntity *specified* in the find_relatedBusinesses, this structure reports complete business relationships with other businessEntity registrations. Business relationships are complete between two businessEntity registrations when the publishers controlling each of the businessEntity structures involved in the relationship set assertions affirming that relationship.

Each relatedBusinessInfo structure contains information about a businessEntity that *relates* to the specified businessEntity by at least one relationship. This information about the related businessEntity comprises its businessKey, name and optional description data, along with a collection element named sharedRelationships that in turn can contain zero or more keyedReference elements. These keyedReference elements, together with the businessKey elements for specified and the related businessEntity represent the complete relationships, that is, matching publisher assertions made by the publishers for each businessEntity. Since the related businessEntity can occupy either side of a relationship, the sharedRelationships element is adorned with a direction attribute. In the example above, Matt's Garage is the toKey, that is, the child of the parent-child relationship to the business with the key "F5E65…".

12.10 serviceDetail

This message returns one or more complete businessService structures in response to a get_serviceDetail inquiry message.

12.10.1 Sample

```
<serviceDetail generic="2.0" operator="uddi.sourceOperator" [truncated="false"]
                xmlns="urn:uddi-org:api_v2">
        <businessService businessKey="F5E65…" serviceKey="3D21…">
                …
        </businessService>
        [<businessService/>…]
</serviceDetail>
```

One can use serviceDetail messages to get complete descriptive and technical details about registered services by providing one or more serviceKey values in the get_serviceDetail message. Notice that the businessKey value is expressed in this message because the container does not provide a link to the parent businessEntity structure.

12.11 serviceList

This message returns zero or more serviceInfo structures in response to a find_service inquiry message.

12.11.1 Sample

```
<serviceList generic="2.0" operator="uddi.sourceOperator" [truncated="false"]
                xmlns="urn:uddi-org:api_v2">
        <serviceInfos>
                <serviceInfo serviceKey="3D45…" businessKey="2E4C…">
                        <name>Purchase Orders</name>
                </serviceInfo>
        </serviceInfos>
</serviceList>
```

ServiceInfo structures are abbreviated versions of businessService data, suitable for populating a list of services associated with a business and that match a pattern as specified in the inputs to the find_service message. Notice that the businessKey attribute is expressed in the serviceInfo elements found in this message. This is because this information is not available from a containing element.

Since a serviceInfo structure can represent a projection to a deleted businessService, the name element within the serviceInfo structure is optional (see section 4.4.13.3 of the V2 API Specification on deleting projected services).

12.12 tModelDetail

This message returns one or more complete tModel structures in response to a get_tModelDetail inquiry message.

12.12.1 Sample

```
<tModelDetail generic="2.0" operator="uddi.sourceOperator" [truncated="false"]
                xmlns="urn:uddi-org:api_v2">
        <tModel tModelKey="uuid:F5E65…" authorizedName="J. Doe"
operator="uddi.publishingOperator" >
                …
        </tModel>
        [<tModel/>…]
</tModelDetail>
```

Because tModel structures are top-level data (that is, stand alone with no parent containers) the authorizedName value is expressed. This is the name of the person whose account was used to register the data. The two operator attributes each express the distinguished name of the Operator Site that is providing the data and the operator where the data is managed.

12.13 tModelList

This message returns zero or more tModelInfo structures in response to a find_tModel inquiry message.

12.13.1 Sample

```
<tModelList generic="2.0" operator="uddi.sourceOperator" [truncated="false"
                xmlns="urn:uddi-org:api_v2">
        <tModelInfos>
                <tModelInfo tModelKey="uuid:34D5…">
                        <name>Proprietary XML purchase order</name>
                </tModelInfo>
                [<tModelInfo/>…]
        </tModelInfos>
</tModelList>
```

The tModelInfo structures are abbreviated versions of tModel data, suitable for finding candidate tModels, populating lists of results and then providing drill-down features that rely on the get_xxDetail messages.

13 Appendix D: Data Field Lengths

The following table summarizes all known stored element and attribute names based on the names of the fields defined in the XML schema. These are the storage length limits for information that is saved in the UDDI registry, given in Unicode characters. The Operator Sites will truncate data that exceeds these lengths. Fields that are generated by the Operator site (ignored on input) are not listed. Keys are listed even though they are generated. Since keys are referenced by other structures, they are shown here.

Field Name	Length
accessPoint	255
addressLine	80
authInfo	4096
authorizedName	255
bindingKey	41
businessKey	41
description	255
discoveryURL	255
email	255
fromKey	41
hostingRedirector	41
instanceParms	255
keyName	255
keyType	16
keyValue	255
name	255
overviewURL	255
personName	255
phone	50
serviceKey	41
sortCode	10
tModelKey	255
toKey	41
uploadRegister	255
URLType	16
useType	255

14 Appendix E: Structured Address Example

The address structure, contained in the businessEntity structure, contains a simple list of addressLine elements. While this is useful for publishing addresses in a UDDI registry or simply printing them on paper, the address' structure and meaning remains hidden for a given businessEntity. For this reason, address structures can be adorned virtually with keyedReference elements. In fact a tModelKey attribute can be provided for an address structure and keyName keyValue attribute pairs can be provided for each addressLine element. This example is provided to demonstrate how the application of tModelKey, keyName and keyValue attributes to address structures can be used to give structure and meaning to a given address.

Let us assume that a community of several country-specific postal agencies, called "IBCPA", not existing in reality, agreed on a core set of address components for exchanging data electronically. This set currently comprises the components Street, Street number, Postal code, City, District, Region and Country.

In order to make these address components available for their use in UDDI address structures, IBCPA assigns a unique value (10, 20, ..., 70) to each address component and publishes a tModel with a save_tModel message call that contains a tModel structure in the following form.

```
<tModel>
        <name>IBCPA.org:address:1.0</name>
        <description xml:lang="en">Codes for Address Components defined by the International
                Board of Postal Agencies</description>
        <overviewDoc>http://www.ibcpa.org/address/codes.html</overviewDoc>
        …
</tModel>
```

IBCPA gets back the tModelKey A548.... As a result, the IBCPA set of address components can now be used by every publisher to structure their addresses in businessEntity structures. The following example shows an address structure using the IBCPA tModel in a save_business message call.

```
<address useType="Sales office" tModelKey="uuid:A548…">
        <addressLine keyName="Street" keyValue="10">Alexanderplatz</addressLine>
        <addressLine keyName="Street number" keyValue="20">12</addressLine>
        …
        <addressLine keyName="Country" keyValue="70">Deutschland</addressLine>
</address>
```